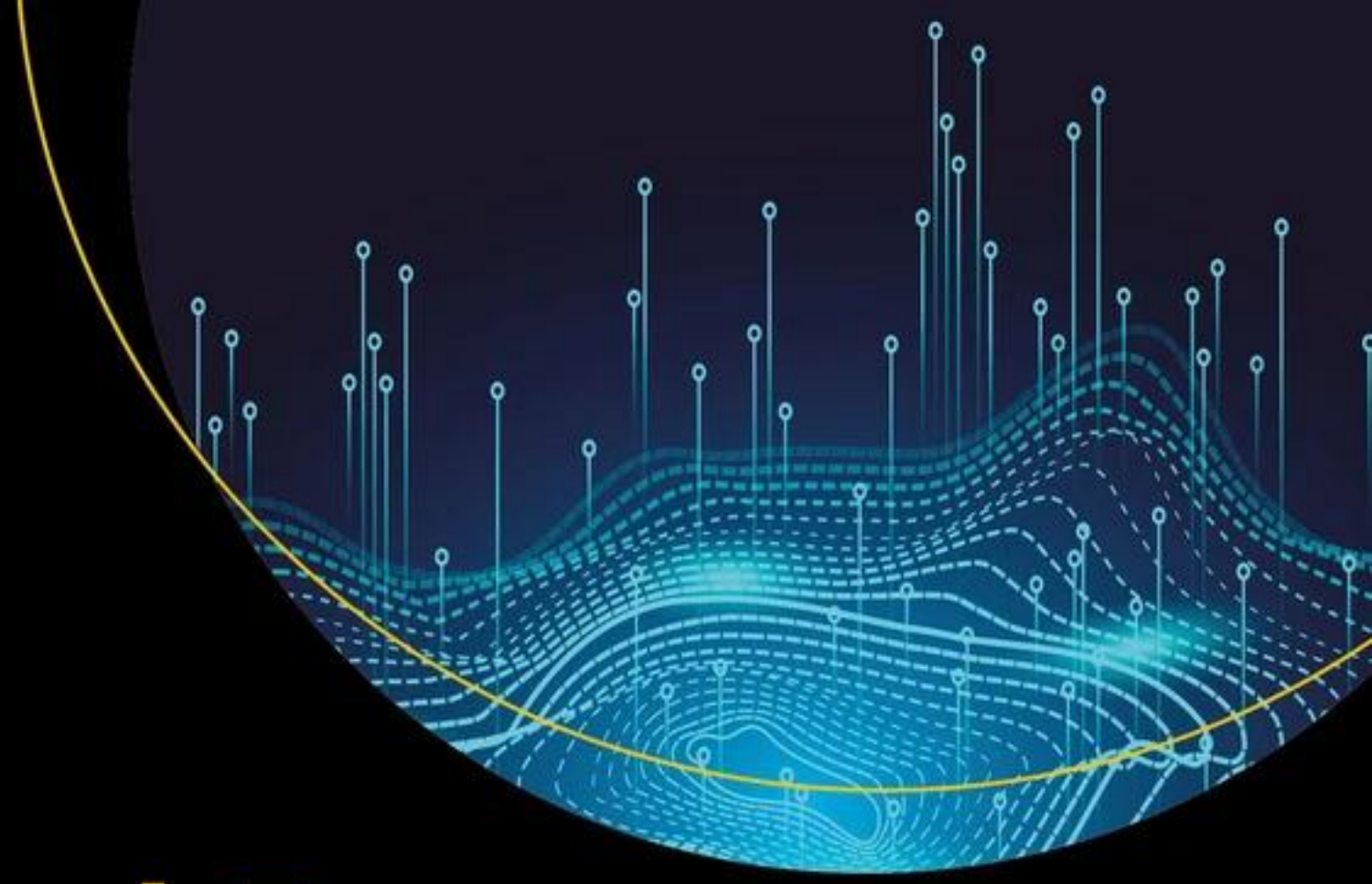


Applied Data Science Using PySpark

Learn the End-to-End Predictive Model-Building Cycle

Second Edition

Ramcharan Kakarla
Sundar Krishnan
Balaji Dhamodharan
Venkata Gunnu

Apress®

Applied Data Science Using PySpark

Learn the End-to-End Predictive Model-Building Cycle

Second Edition

Ramcharan Kakarla
Sundar Krishnan
Balaji Dhamodharan
Venkata Gunnu

Apress®

Applied Data Science Using PySpark: Learn the End-to-End Predictive Model-Building Cycle, Second Edition

Ramcharan Kakarla
Philadelphia, PA, USA

Sundar Krishnan
Philadelphia, PA, USA

Balaji Dhamodharan
Texas, TX, USA

Venkata Gunnu
NJ, USA

ISBN-13 (pbk): 979-8-8688-0819-7
ISBN-13 (electronic): 979-8-8688-0820-3
https://doi.org/10.1007/979-8-8688-0820-3

Copyright © 2024 by Ramcharan Kakarla, Sundar Krishnan, Balaji Dhamodharan and Venkata Gunnu

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Aditee Mirashi
Development Editor: James Markham
Coordinating Editor: Kripa Joseph
Copy Editor: Kezia Endsley

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (https://github.com/Apress). For more detailed information, please visit https://www.apress.com/gp/services/source-code.

If disposing of this product, please recycle the paper

Table of Contents

About the Authors xi

About the Technical Reviewer xiii

Acknowledgments xv

Introduction xvii

Chapter 1: Setting Up the PySpark Environment 1

Local Installation Using Anaconda 1

Install Anaconda 1

Conda Environment Creation 3

Download and Unpack Apache Spark 5

Install Java 8 or Later 5

Mac and Linux Users 6

Windows Users 6

Run PySpark 9

Jupyter Notebook Extension 10

Docker-Based Installation 11

Why Do We Need to Use Docker? 11

What Is Docker? 12

Create a Simple Docker Image 13

Download PySpark Docker 14

Step-by-Step Approach to Understanding the PySpark Docker Commands 15

Databricks Community Edition 17

Create a Databricks Account 19

Create a New Cluster 19

Create Notebooks 20

Import Data Files into the Databricks Environment 21

GitHub Codespaces 23
Basic Operations 28
Upload Data 28
Access Data 29
Calculate Pi 29
Summary 30

Chapter 2: PySpark Basics 31

PySpark Background 31
PySpark Resilient Distributed Datasets (RDDs) and DataFrames 35
Data Manipulations 39
Reading Data from a File 39
Reading Data from a Hive Table 40
Reading Metadata 40
Counting Records 41
Subset Columns and View a Glimpse of the Data 42
Missing Values 43
One-Way Frequencies 44
Sorting and Filtering One-Way Frequencies 46
Casting Variables 47
Descriptive Statistics 50
Unique/Distinct Values and Counts 51
Filtering 53
Creating New Columns 56
Deleting and Renaming Columns 61
Pandas API on Spark 61
Summary 66

Chapter 3: Utility Functions and Visualizations 67

Additional Data Manipulations 67
String Functions 67
Registering DataFrames 69

Window Functions 70
Other Useful Functions 75
Data Visualizations 88
Introduction to Machine Learning 93
Summary 98

Chapter 4: Variable Selection 99

Exploratory Data Analysis 101
Cardinality 101
Missing Values 102
Built-in Variable Selection Process: Without Target 114
Principal Component Analysis (PCA) 115
Singular Value Decomposition (SVD) 123
Built-in Variable Selection Process: With Target 124
ChiSq Selector 125
Model-based Feature Selection 128
Custom-built Variable Selection Process 133
Information Value Using Weight of Evidence 133
Custom Transformers 143
Voting-based Selection 149
Summary 152

Chapter 5: Supervised Learning Algorithms 153

Machine Learning Algorithm Basics 153
Regression 155
Classification 155
Loss Functions 155
Optimizers 158
Gradient Descent 160
Stochastic/Mini-batch Gradient Descent 163
Momentum 164
AdaGrad (Adaptive Gradient) Optimizer 165

Root Mean Square Propagation (RMSprop) Optimizer 166
Adaptive Moment (Adam) Optimizer 166
Activation Functions 167
Linear Activation Function 168
Batch Normalization 173
Dropout 173
Supervised Machine Learning Algorithms 174
Linear Regression 174
Logistic Regression 180
Decision Trees 187
Random Forests 198
Gradient Boosting 201
Support Vector Machine (SVM) 203
Neural Networks 205
One-vs-Rest Classifier 211
Naïve Bayes Classifier 213
Regularization 216
Summary 217

Chapter 6: Model Evaluation 219

Model Complexity 219
Underfitting 220
Best Fitting 220
Overfitting 221
Bias and Variance 222
Model Validation 223
Training/Test Split 223
k-fold Cross-Validation 228
Leave-One-Out Cross-Validation 230
Leave-One-Group-Out Cross-Validation 230
Time-series Model Validation 233

Leakage 233
Target Leakage 233
Data Leakage 234
Issues with Leakage 234
Model Assessment 235
Continuous Target 235
Binary Target 241
Summary 262

Chapter 7: Unsupervised Learning and Recommendation Algorithms 263

Segmentation 263
Distance Measures 266
Types of Clustering 274
Latent Dirichlet Allocation (LDA) 290
LDA Implementation 294
Collaborative Filtering 302
Matrix Factorization 304
Summary 308

Chapter 8: Machine Learning Flow and Automated Pipelines 311

MLflow 312
MLflow Code Setup and Installation 313
MLflow User Interface Demonstration 317
Automated Machine Learning Pipelines 323
Pipeline Requirements and Framework 324
Data Manipulations 327
Feature Selection 327
Model Building 328
Metrics Calculation 328
Validation and Plot Generation 328
Model Selection 328
Score Code Creation 328

Collating Results 329
Framework 329
Pipeline Outputs 331
Summary 334

Chapter 9: Deploying Machine Learning Models 335

Starter Code 335
Save Model Objects and Create Score Code 335
Model Objects 336
Score Code 336
Model Deployment Using HDFS Object and Pickle Files 337
Model Deployment Using Docker 338
The requirements.txt File 339
Dockerfile 340
Changes Made in the helper.py and run.py Files 341
Create Docker and Execute Score Code 341
Real-Time Scoring API 342
Postman API 343
Test Real-Time Using Postman API 346
Build UI 346
The streamlitapi Directory 347
real_time_scoring Directory 348
Executing the docker-compose.yml File 348
Real-time Scoring API 349
Summary 351

Chapter 10: Experimentation and Optimization Strategies 353

Importance of Experimentation 353
The Experimentation Process 354
Hypothesis Testing 355
Experimentation Case Study 356

Optimization Strategies with PySpark 363
Random Data Generation 363
Sampling 364
Simple Random Sampling (SRS) 364
Stratified Sampling 365
Difference Between the coalesce and repartition Functions 365
Switching between Python and PySpark 366
Curious Character of Nulls 367
Common Function Conflicts 367
Join Conditions 367
User Defined Functions (UDFs) 368
Handle the Skewness 368
Using Cache 369
Persist/Unpersist 369
Shuffle Partitions 369
Use Optimal Formats 370
Data Serialization 370
Accomplishments 371

Chapter 11: Modeling Frameworks 373

Customer Lifetime Value 373
Understanding Customer Lifetime Value (CLV) 374
Importance of CLV 374
Types of CLV 376
CLV in Different Business Models 378
Challenges and Considerations in Implementing CLV 381
Best Practices for Leveraging CLV 383
Types of CLV Models 385
Return on Marketing Investment (ROMI) 406
The Basic Formula for ROMI 407
Step-by-Step Approach to Calculating ROMI Using Data Science Models 407
Advanced Techniques and Considerations 412

Machine Learning for Predictive Modeling 413
Real-Time Data Processing 414
Experimentation and A/B Testing 414
ROMI Conclusion 415
Product Engagement Scores 416
Importance of Product Engagement Scores 416
Example 1: Streaming Platform 417
Example 2: Software Application 419
Importance of Product Engagement Scores 422
Uplift Modeling 422
Importance of Uplift Modeling 423
The Theory of Uplift Modeling 423
Basic Steps in Uplift Modeling 423
Using the Uplift Quadrant Matrix 425
Summary 433
Conclusion 434

Index 435

About the Authors

Ramcharan Kakarla is currently the Principal ML at Altice USA. He is a passionate data science and artificial intelligence advocate with ten years of experience. He holds a master's degree from Oklahoma State University, specializing in data mining. He is also currently pursuing a master's degree in management from the University of California, LA. Prior to UCLA and OSU, Ramcharan received his bachelor's in electrical and electronics engineering from Sastra University in India. He was born and raised in the coastal town of Kakinada, India and started his career working as a performance engineer with several Fortune 500 clients, including State Farm, British Airways, Comcast, and JP Morgan Chase. In his current role, he is focused on building Generative AI, data science solutions, and frameworks leveraging big data. He has published several papers and posters in the field of predictive analytics. He also served as SAS Global Ambassador in 2015.

Sundar Krishnan is a Senior Data Science Manager at CVS Health. He has more than 12 years of extensive experience leading cross-functional data science teams and is an AI, ML, and cloud platform expert. He has a proven track record of building high-performing teams and implementing innovative AI strategies to optimize operational costs and generate substantial revenue. An expert in 0 to 1 product development, Sundar successfully led teams from conception to market-ready products in GenAI and data science. Sundar was born and raised in Tamil Nadu, India, and has a bachelor's degree from the Government College of Technology, Coimbatore. He completed his master's degree at Oklahoma State University, Stillwater. He blogs about his data science works on Medium in his spare time.

Balaji Dhamodharan is an award-winning global data science leader, guiding teams to develop and implement innovative, scalable ML solutions. He currently leads the AI/ML and MLOps strategy initiatives with NXP semiconductors. He has over a decade of experience delivering large-scale technology solutions across diverse industries. His expertise spans software engineering, enterprise AI platforms, AutoML, MLOps, and Generative AI technologies. Balaji holds Masters' degrees in management information systems and data science from Oklahoma State University and Indiana University. Originally from Chennai, India, Balaji currently resides in Austin, TX.

Venkata Gunnu is a senior executive director of knowledge management and innovation at JPM Chase. He is an executive with a successful background crafting enterprise-wide data and data science solutions, GenAI, process improvements, and data and data science-centric products. Venkata is a concept-to-implementation strategist with demonstrated success controlling multiple projects that elevate organizational efficiency while optimizing resources. He is data-focused and analytical, with a track record of automating functions, standardizing data management protocol, and introducing new business intelligence solutions.

About the Technical Reviewer

Anvesh Reddy Minukuri currently serves as senior lead VP of Generative AI/ML engineering at JPMorgan Chase, specializing in NLP applications. With over 11 years in IT and nine years in analytics, he has a robust background in developing predictive and prescriptive solutions. Anvesh holds a master's degree in data mining from Oklahoma State University and a bachelor's degree in computer science from JNTU University, India.

Originating from southern India, he began his career as a software engineer, working with prestigious fortune 500 clients such as GE, Cisco, and Tech Mahindra. Anvesh has been instrumental in helping stakeholders leverage AI and ML for actionable data insights and has overseen the design of machine learning solutions. His expertise includes crafting data science solutions and ML pipeline frameworks to address predictive, prescriptive, and NLP challenges. Proficient in LLM model development, assessment, and optimization, he excels in NLP preprocessing, data storage, infrastructure management, and scaling.

Anvesh has significantly contributed to the field of AI/ML through various published papers and posters and has driven adoption across diverse NLP applications.

Acknowledgments

This book could not have been written without the participation and assistance of many people. I am deeply indebted and appreciate all the support that I received from my friends, family, and colleagues during the tough times. Their contributions are sincerely appreciated.

I want to express my deepest appreciation to Dr. Goutam Chakraborty, Director, MS in business analytics, OSU, for introducing me to the world of data science a decade ago. His passion was contagious, and this book truly is a product of that passion.

I also want to thank my parents, Mr. Mohan Rao Kakarla and Mrs. Venkata Rani, for their enthusiasm and motivation. I am also very grateful to my life partner Mrs. Prasanthi and my daughter Arha Kakarla who supported me at all times. I want to thank my friend Mr. Sundar for being a great co-author. I am very thankful to Mr. Subba Rao Mulakala for being a great mentor and supporter as well. Thank you to all my friends and family who supported me during this journey.

—Ramcharan Kakarla

Introduction

This comprehensive guide, with hand-picked examples of daily use cases, walks you through the end-to-end predictive model-building cycle, including the latest techniques and tricks of the trade. The first three chapters start by setting up the environment, covering the basics of PySpark, and focusing on data manipulation. Feature engineering is where data science professionals spend 70 percent of their time. As we make you comfortable with the language, we build upon that to introduce you to the mathematical functions available off the shelf. Before you move to the next chapter, you learn about the predictive modeling framework.

Chapter 4 dives into the art of variable selection. It demonstrates various selection techniques available in PySpark. Chapters 5, 6, and 7 take you on the journey of machine learning algorithms, including implementation and fine-tuning techniques. In addition, those chapters explain different validation metrics and how to use them to pick the best models. Chapters 8 and 9 walk you through various machine learning pipelines, including methods available to operationalize the model and serve it through Docker/API. Chapter 10 includes some tricks that can help you optimize your programs and machine learning pipelines. In Chapter 11, you learn about frameworks you can use to unlock complete value from your data. In the end, you will be comfortable using PySpark in data science applications.

This book is recommended for data science enthusiasts who want to unleash the power of parallel computing by working with big datasets. It's also highly recommended for professionals who want to switch from traditional languages to open-source in a big data setting. It's a value-add for students who want to work on big data as well.

What you will learn:

- Overview of end-to-end predictive model building
- Multiple variable selection techniques and implementations
- Multiple algorithms and implementations
- Operationalizing models

- Data science experimentations and tips
- Modeling frameworks

This book includes:

- A comprehensive guide to data science solutions in PySpark
- Specially designed exercises to help you practice your knowledge
- Industry standard methods and procedures, implemented with examples
- An easy-to-follow structure for a broad range of audiences (from professionals to students)
- Advice for transitioning your data science solutions from traditional languages to PySpark
- Handpicked tips and tricks that can help you with your day-to-day work

CHAPTER 1

Setting Up the PySpark Environment

This chapter is dedicated to setting up the PySpark environment. We discuss multiple options, and you can pick your favorite. For those with an environment already set up, feel free to skip ahead to the "Basic Operations" section later in this chapter.

This chapter covers the following topics:

- Local installation using Anaconda
- Docker-based installation
- Databricks Community Edition
- GitHub codespaces

Local Installation Using Anaconda

Install Anaconda

The initial step is to download Anaconda at https://www.anaconda.com/download. You can download the individual edition at no cost. We suggest using the latest version of Python (Python 3.11), as we utilize this version in all the examples. If you have an alternate version installed, adjust your code accordingly. After installing Anaconda, you can execute the following command in the Terminal/Anaconda prompt to confirm that the installation has been successfully completed.

Note Windows users should use the Anaconda prompt, not the Command prompt. This option will be available after you install Anaconda.

© Ramcharan Kakarla, Sundar Krishnan, Balaji Dhamodharan and Venkata Gunnu 2024
R. Kakarla et al., *Applied Data Science Using PySpark*, https://doi.org/10.1007/979-8-8688-0820-3_1

conda info

The output should look like the following.

Output Shown in Terminal/Anaconda Prompt

```
(base) sundar@Sundars-MacBook-Pro / % conda info

     active environment : base
    active env location : /Users/sundar/anaconda3
            shell level : 1
       user config file : /Users/sundar/.condarc
 populated config files : /Users/sundar/.condarc
          conda version : 23.7.4
    conda-build version : 3.26.1
         python version : 3.11.5.final.0
       virtual packages : __archspec=1=x86_64
                          __osx=10.16=0
                          __unix=0=0
       base environment : /Users/sundar/anaconda3  (writable)
      conda av data dir : /Users/sundar/anaconda3/etc/conda
  conda av metadata url : None
          channel URLs : https://repo.anaconda.com/pkgs/main/osx-64
                         https://repo.anaconda.com/pkgs/main/noarch
                         https://repo.anaconda.com/pkgs/r/osx-64
                         https://repo.anaconda.com/pkgs/r/noarch
         package cache : /Users/sundar/anaconda3/pkgs
                         /Users/sundar/.conda/pkgs
       envs directories : /Users/sundar/anaconda3/envs
                         /Users/sundar/.conda/envs
               platform : osx-64
             user-agent : conda/23.7.4 requests/2.31.0 CPython/3.11.5
                         Darwin/21.6.0 OSX/10.16 aau/0.4.2
                         c/1VatYgfffAMMKXQOgPBo-Q s/nX_yel1PDom6K2c64Czdcg
                         e/u9Ki7wvzxHtwkihYPDu5wA
                UID:GID : 501:20
             netrc file : /Users/sundar/.netrc
           offline mode : False
```

Note Every conda operation starts with the keyword conda.

For more information on conda operations, search the web for "Conda cheat sheet" and follow the documentation provided here: https://docs.conda.io/.

Conda Environment Creation

In general, programming languages have various versions. When you create a program on a specific version and attempt to run it on another version, the chances of a successful execution depend on the differences between the two versions. This might be a challenging experience. Programs developed on an older version may error out on a newer version because of these dependencies. This is where a virtual environment becomes useful. These environments isolate dependencies into separate sandboxes, making it easier to run programs or applications in the corresponding environments. The Anaconda tool is an open-source platform where you can manage and run your Python programs and environments. You can create multiple environments with different versions and switch between environments as needed. It is good practice to determine which environment you are currently in before executing any program or application.

We want to focus on a specific command in the conda cheat sheet for environment creation. The syntax is provided here:

```
conda create --name ENVNAME python=3.11
```

Before creating the environment, use another Conda command to list your existing environments.

```
conda env list
```

Once you run this command, you get the following output.

Environment List Before Creation:

```
# conda environments:
#
base                  *  /Users/sundar/anaconda3
```

The base environment exists by default in Anaconda. The base is the active environment, represented by a small asterisk * symbol to its right. This means that whatever conda operations you perform (install, update, or removal) will be carried out in the default base environment.

Now that you understand what the base environment is, you can create your own PySpark environment. In this example, we replace ENVNAME with a name (pyspark_env) that we want to call the environment.

```
conda create --name pyspark_env python=3.11
```

When it prompts for user input, type y and press Enter. It is good practice to type this code rather than copying it from the text since a missing hyphen can create an error, such as "CondaValueError: The target prefix is the base prefix. Aborting" or "conda: error: unrecognized arguments: --name." After executing the command, perform the conda environment list again. This time, you get a slightly different output.

Conda Environment Right After Creation

```
# conda environments:
#
base                  *  /Users/sundar/anaconda3
pyspark_env              /Users/sundar/anaconda3/envs/pyspark_env
```

Notice that a new environment, called pyspark_env, has been created. Still, base is the active environment. You can activate the new environment, pyspark_env, by using the following command.

Conda Environment After the Change

```
conda activate pyspark_env
```

```
# conda environments:
#
base                     /Users/sundar/anaconda3
pyspark_env           *  /Users/sundar/anaconda3/envs/pyspark_env
```

Notice that the * indicates the current environment. From now on, all the conda operations will be performed in the new PySpark environment. The next section explains the Spark installation process.

Download and Unpack Apache Spark

You can download the latest version of Apache Spark at https://spark.apache.org/downloads.html. This book uses Spark 3.5.0. (Specific examples in the book might use other Spark versions, but for the most part, all use cases are shown in Spark 3.5.0.) Once you visit the site, you can pick a Spark release and download the appropriate file (see Figure 1-1).

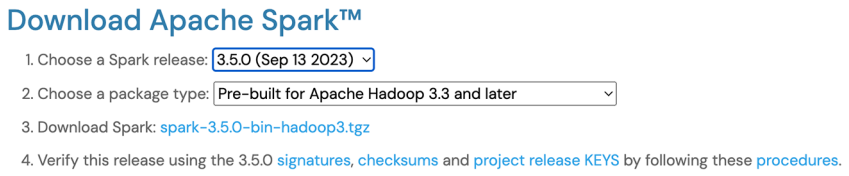

Download Apache Spark™
1. Choose a Spark release: 3.5.0 (Sep 13 2023)
2. Choose a package type: Pre-built for Apache Hadoop 3.3 and later
3. Download Spark: spark-3.5.0-bin-hadoop3.tgz
4. Verify this release using the 3.5.0 signatures, checksums and project release KEYS by following these procedures.

Figure 1-1. *Spark download*

Unzip the file; optionally, you can move it to a specific folder.

Mac/Linux Users

```
tar -xzf spark-3.5.0-bin-hadoop3.tgz
mv spark-3.5.0-bin-hadoop3 /opt/spark-3.5.0
```

Windows Users

```
tar -xzf spark-3.5.0-bin-hadoop3.tgz
move spark-3.5.0-bin-hadoop3 C:\Users\<username>\spark-3.5.0
```

Install Java 8 or Later

You can check for an existing version of Java in your system by typing the following in the Terminal/Anaconda prompt:

```
java -version
```

```
java version "21.0.2" 2024-01-16 LTS
Java(TM) SE Runtime Environment (build 21.0.2+13-LTS-58)
Java HotSpot(TM) 64-Bit Server VM (build 21.0.2+13-LTS-58, mixed mode,
sharing)
```

It's best to use Java version 8 or later. You can always download the latest version at https://www.oracle.com/java/technologies/javase-downloads.html. It is recommended to use Java 8 JDK. When you use a Java version later than 8, you may receive warnings when you launch PySpark. However, these warnings can be ignored.

Mac/Linux users can now proceed to the next section and Windows users can proceed to the section called "Windows Users."

Mac and Linux Users

Assuming you have completed the Spark download, you must now update the `~/.bash_profile` or `~/.bashrc` files to find the Spark installation. This example uses the `bash_profile` file. In the Terminal, type the following code to open the file:

```
vi ~/.bash_profile
```

Once the file comes up, type `i` to insert contents. Add the following lines to set up the environment variables:

```
export SPARK_HOME=/opt/spark-3.5.0
export PATH=$SPARK_HOME/bin:$PATH
export PYSPARK_PYTHON=python3
```

After adding these configurations, press ESC, type `:wq!`, and press Enter to save the file. To apply these changes in the `pyspark_env` environment, run the following command in the Terminal:

```
source ~/.bash_profile
```

When you restart your system, change the `conda` environment to `pyspark_env` to run PySpark.

Windows Users

Assuming you completed the Spark download, you must do a few things to ensure your PySpark installation works as intended. Since you ran the `move` option, your Spark folder should exist in the path provided here:

```
C:\Users\username\spark-3.5.0
```

Navigate to this path via the Anaconda prompt and run the following code:

```
cd C:\Users\username\spark-3.5.0
mkdir hadoop\bin
```

Once the directory is created, you must download the `winutils.exe` file and add it to your Spark installation. Without the `winutils.exe` file, when you invoke PySpark, you might get the following error:

```
"ERROR Shell: Failed to locate the winutils binary in the hadoop binary
path....."
```

In addition, there could be errors in the future when you try to use the `spark-submit` utility. To overcome these errors, download the `winutils.exe` file at https://github.com/cdarlint/winutils. Based on the Hadoop version you installed, you might have to pick the appropriate path in this link. Navigate to the `bin` folder inside the path and click `winutils.exe` to download the file (see Figure 1-2).

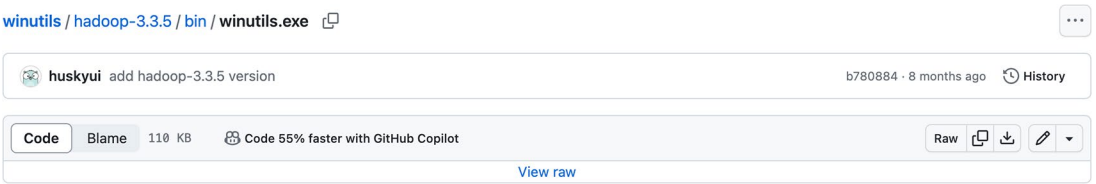


Figure 1-2. *Winutils.exe*

Readers using Hadoop version 3.3 can download the `winutils.exe` file directly from this link: https://github.com/cdarlint/winutils/blob/master/hadoop-3.3.5/bin/winutils.exe. Place the downloaded file in the `hadoop\bin` path, which you created earlier. Alternatively, you can use the following command to perform the same operation:

```
move C:\Users\username\Downloads\winutils.exe C:\Users\username\
spark-3.5.0\hadoop\bin
```

That's it. You are almost done. The final step is to configure the environment variables. You can search for "Environment Variables" in Windows, or you can find them by choosing System Properties ➤ Advanced ➤ Environment Variables. In this example, we add a few variables in the User Variables tab. Click New to see the option shown in Figure 1-3.

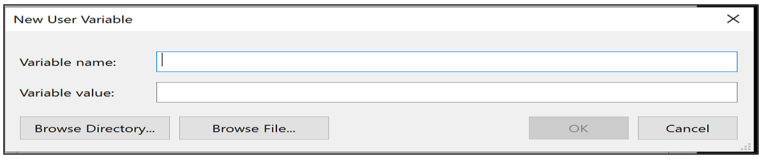


Figure 1-3. *Add a new user variable*

You need to add three things: SPARK_HOME, HADOOP_HOME, and PATH.

SPARK_HOME

```
Variable name - SPARK_HOME
Variable value - C:\Users\username\spark-3.5.0
```

HADOOP_HOME

```
Variable name - HADOOP_HOME
Variable value - C:\Users\username\spark-3.5.0\hadoop
```

When there is an existing PATH variable in the User Variables tab, click Edit and add the following:

```
C:\Users\username\spark-3.5.0\bin
```

That is all. You can now run PySpark. Sometimes, depending on the Spark version, you might need to add two other environment variables—SPARK_LOCAL_HOSTNAME and JAVA_HOME. The settings for these two variables are provided here:

SPARK_LOCAL_HOSTNAME

```
Variable name - SPARK_LOCAL_HOSTNAME
Variable value - localhost
```

JAVA_HOME

```
Variable name - JAVA_HOME
Variable value - C:\Program Files\Java\(your JDK version directory)
```

To apply all these environment changes, restart the Anaconda prompt and change the conda environment to pyspark_env.

Run PySpark

You are all set now. You need to ensure that the conda environment is set to pyspark_env before calling pyspark. Just invoke pyspark from the Terminal/Anaconda prompt. You should see the following pop up.

Run PySpark in the Terminal/Anaconda Prompt

```
Python 3.11.7 (main, Dec 15 2023, 12:09:04) [Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
24/01/19 21:22:55 WARN Utils: Your hostname, Sundars-MacBook-Pro.local
resolves to a loopback address: 127.0.0.1; using 192.168.68.53 instead (on
interface en0)
24/01/19 21:22:55 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to
another address
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
24/01/19 21:22:56 WARN NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.5.0
      /_/

Using Python version 3.11.7 (main, Dec 15 2023 12:09:04)
Spark context Web UI available at http://192.168.68.53:4040
Spark context available as 'sc' (master = local[*], app id =
local-1705720977191).
SparkSession available as 'spark'.
>>>
```

Cool! You can now extend the setup further to work with Jupyter IDE.

Jupyter Notebook Extension

For this, you might have to install Jupyter in your `pyspark_env` environment. You can use the following code to perform the operation.

Mac/Linux Users

```
conda install jupyter
```

You need to add the following two lines of code to your `~/.bash_profile` or `~/.bashrc` setup:

```
export PYSPARK_DRIVER_PYTHON=jupyter
export PYSPARK_DRIVER_PYTHON_OPTS='notebook'
```

Make sure to apply the current changes in the environment by using the corresponding code:

```
source ~/.bash_profile
source ~/.bashrc
```

From now on, whenever you invoke `pyspark`, it should automatically invoke a Jupyter Notebook session. If the browser does not open automatically, you can copy the link and paste it into the browser.

Jupyter Invocation Links

```
Jupyter Server 2.10.0 is running at:
[I 2024-01-19 21:55:24.577 ServerApp] http://localhost:8888/tree?token
=b365126767afe1edb9d2ccddf00e203e1d3808b031e1a807
[I 2024-01-19 21:55:24.577 ServerApp]     http://127.0.0.1:8888/tree?token
=b365126767afe1edb9d2ccddf00e203e1d3808b031e1a807
```

Windows Users

You need to install a package using the Anaconda prompt.

```
python -m pip install findspark
```

From now on, when you invoke `pyspark` from the Anaconda prompt, it should automatically invoke a Jupyter Notebook session. If the browser does not open automatically, you can copy the link and paste it into the browser. In the browser, you need to select a New Python 3 notebook. To test the PySpark installation in Windows, run the following code in Jupyter.

```
## Initial Check for Spark installation
import findspark
findspark.init()

## Run these codes when the initial check is successful
## Create a Spark session and test it using spark.sql

import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
df = spark.sql("select 'spark' as hello ")
df.show()
```

Well, that is all with the setup. You can skip to the "Basic Operations" section later in this chapter.

Docker-Based Installation

Believe it or not, a Docker-based installation is effortless compared to a local installation. Some readers may be just starting with Docker, whereas others might be proficient. This section introduces Docker to readers new to this topic and shows you how to use PySpark with Docker. We want to clarify before going further: Docker is a vast resource, and this book cannot cover it all. You can read online articles or buy a separate book that teaches you about Docker. Here, we cover the basic concepts necessary for using Docker. Proficient users can skip to the "Create a Simple Docker Image" section in this chapter.

Why Do We Need to Use Docker?

Imagine a scenario where you developed an application using Python 2.7. Suddenly, your college or company administration announces a switch to Python 3.11 or a later version. You are faced with a challenge to test your code's compatibility with the new Python version and update your code when it errors out. Given the time required to ensure your application runs smoothly, this process can be frustrating and cumbersome.

In such situations, Docker proves to be incredibly useful. Once you create a Docker image, changes occurring in the external environment become irrelevant. Your application will function seamlessly wherever Docker is deployed. This feature makes Docker exceptionally powerful and a vital tool for data science professionals.

As a data scientist, you may encounter scenarios where you need to develop and deploy production models. Even a minor change in the production environment has the potential to disrupt your code and application. Therefore, leveraging Docker simplifies your job and ensures smoother operations.

What Is Docker?

Docker is a platform-as-a-service (PaaS) product. You create the platform to run your application. Once the platform is built, it can be hosted anywhere you have Docker. Docker operates on various host operating systems, including Mac/Windows/Linux, and so on. After installing the Docker application, you can create Docker images, which serve as templates for building containers and an isolated system environment. Inside a container, you host your application and install all its dependencies (*bins/libs*). This setup is shown in Figure 1-4.

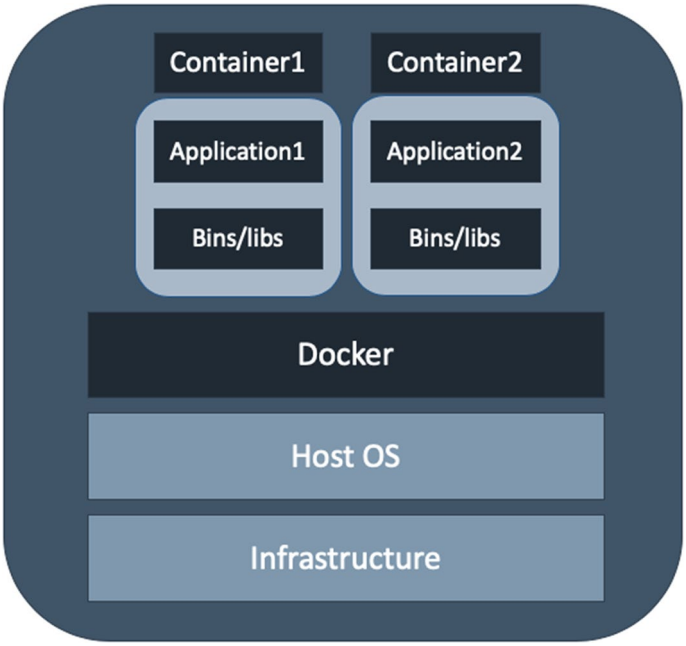


Figure 1-4. *Docker*

Wait a second! It sounds like a virtual machine. Well, it is not. Docker differs in its internal working mechanism. While each virtual machine requires its own operating system, Docker allows the operating system to be shared. Additionally, virtual machines consume more disk space and have higher overhead utilization compared to Docker. In the next step, you will create a simple Docker image.

Create a Simple Docker Image

Before you create an image, open a Docker account as follows:

1. Create a Docker Hub account at https://hub.docker.com/signup.
2. Download Docker Desktop from https://www.docker.com/get-started.
3. Install Docker Desktop and sign in using the Docker Hub user ID/password.

To create a Docker image, you need a *Dockerfile*. The Dockerfile commands are available here: https://docs.docker.com/engine/reference/builder/. To try it out, you will see how to create a basic Python image and run an application inside it. Using a text editor, open a new file named `Dockerfile` and copy/paste the following commands:

```
FROM python:3.11-slim
COPY test_script.py /
CMD ["python", "./test_script.py"]
```

The `test_script.py` file has the following commands. This file should exist in the same location as the `Dockerfile` for this code to work:

```
import time
print("Hello world!")
print("The current time is - ", time.time())
```

What is happening here? You are using a base image, which is Python 3.11, a slim version. Next, you copy the local file called `test_script.py` to Docker. Finally, you execute the `test_script.py` file. That's all. You can now create the Docker image using Docker commands. All the Docker commands are provided here: https://docs.docker.com/engine/reference/commandline/docker/. For this simple image creation, we will look at these two commands:

- `docker build`: Create an image
- `docker run`: Run an image

From your Terminal/Command prompt, navigate to the folder where `Dockerfile` resides and run the following command:

```
# Build the Docker image with the specified tag
docker build -t username/test_image .
# Run a Docker container
docker run username/test_image
```

The first command creates the image. The `-t` is used to assign a tag to the image. You can replace `username` with your actual Docker Hub username. Make sure to include the `period (.)` after the tag name. This will automatically consume the `Dockerfile` in the current location to create the image. The second command runs the image. When executed, you should get an output saying, `"Hello, world! The current time is`– blah. blah."

You should now have a basic understanding of Docker, so next you download PySpark Docker.

Download PySpark Docker

Follow these steps to download PySpark Docker:

1. Go to `https://hub.docker.com/` and search for the PySpark image.
2. Navigate to the image shown in Figure 1-5.

Figure 1-5. *PySpark image*

3. From the Terminal/Anaconda prompt, type the following code and press Enter:

   ```
   docker pull jupyter/pyspark-notebook
   ```

4. Once the image is completely downloaded, you can run the following code to ensure the image exists:

   ```
   docker images
   ```

5. It's worth noting that "latest" tag is applied to the image. This tag serves to distinguish between different versions of the image with the same name.

Step-by-Step Approach to Understanding the PySpark Docker Commands

1. Start with the basic Docker `run` command, as shown in Listing 1-1.

   Listing 1-1. Docker Run Base

   ```
   docker run jupyter/pyspark-notebook:latest
   [I 2024-01-21 19:09:27.110 ServerApp]
   http://79a9222b33d3:8888/lab?token=7b62b7b9c7f1aa378a6d1ebbd
   ecc54cafbfcb877844e7ed9
   [I 2024-01-21 19:09:27.110
   ServerApp]     http://127.0.0.1:8888/lab?token=7b62b7b9c7f1
   aa378a6d1ebbdecc54cafbfcb877844e7ed9
   ```

   After invoking this command, you'll see a few lines of output, and a link appears, as shown.

   When you copy and paste the preceding link in the browser, you won't see a Jupyter session. This is because no connection is established between Docker and the outside world. In the next step, you will set the connection by configuring a port.

2. You can establish a port with the following command:

   ```
   docker run -p 7777:8888 jupyter/pyspark-notebook:latest
   ```

   We've selected two different ports here solely for explanatory purposes. After you execute this command, you will have a similar output as that shown in Listing 1-1. You won't see any session when you copy and paste

as before. However, instead of using port 8888, it will work when you use the port 7777. Try it with this link: `http://localhost:7777/`. You should be able to see a Jupyter session. What is happening here?

Inside Docker, the Jupyter Notebook session is running in port 8888. However, when we established the Docker connection, we explicitly mentioned that Docker should use port 7777 to communicate with the outside world. Port 7777 becomes the doorway, so we see the output. Now that you are clear about the port syntax, you can assign the same port to both sides as follows. This time, the link shown should work directly.

```
docker run -p 8888:8888 jupyter/pyspark-notebook:latest
```

3. What if you want to access external files within Docker? This is again established using a simple command:

```
docker run -p 8888:8888 -v /Users/sundar/Desktop/Applied_
DS_Second_Edition/: /home/jovyan/work/ jupyter/pyspark-
notebook:latest
```

Let's examine this command in depth. As with the port mapping you saw earlier, notice the symbol `-v` followed by the paths separated by `:` (a colon). The `-v` denotes the volume to mount while invoking the `docker` command. The left side is the local directory (`/Users/sundar/Desktop/Applied_DS_Second_Edition/`), and the right is the directory in Docker (`/home/jovyan/work`). In the next step, you see how to invoke the Docker container in interactive mode.

4. How do you invoke in an interactive/bash mode? Use the following command:

```
docker run -it -p 8888:8888 -v /Users/sundar/Desktop/
Applied_DS_Second_Edition/:/home/jovyan/work/ jupyter/pyspark-
notebook:latest bash
```

Observe two changes in this command. First, the addition of the `-it` tag indicates the initiation of an interactive session. Second, the inclusion of the bash command triggers the shell. Upon executing

this command, you won't encounter a Jupyter session link. Instead, you enter a Linux shell environment, allowing you to execute additional commands. This functionality proves useful in various scenarios. For instance, you can install required packages within the Linux environment. Navigate to the previously mapped folder and initiate the Jupyter Notebook using the following command:

```
cd /home/joyvan/work
jupyter notebook
```

You should be able to see the files and folders in the Jupyter session.

5. There are additional tags for space optimization. We recommend you use the following final command when you run Docker:

```
docker run -it --rm --shm-size=1g --ulimit memlock=-1 -p 8888:8888
-v /Users/sundar/Desktop/Applied_DS_Second_Edition/:/home/jovyan/
work/ jupyter/pyspark-notebook:latest
```

The `--rm` tag deletes the container after completing the specified task. This is suitable for short-lived containers. You explicitly tell the Docker daemon that once this container is finished running, it should clean up the files and delete them to save disk space. The second tag, `--shm-size,` is used to increase/decrease the shared memory size. For memory-intensive containers, you can utilize this option to specify the shared memory size, which helps the containers run faster by giving additional access to allocated memory. The final tag is `--ulimit memlock=-1`, and it provides the container with an unlimited locked-in-memory address space. This will lock the maximum available shared memory during the container invocation. You don't have to use all these options; however, it is good to know them. Now, you have a working PySpark session with Docker.

Databricks Community Edition

This is intended for users who are looking to use Databricks. If you already have the Databricks environment at your workplace, you can contact your admin to create an account. For readers who intend to install this, use this link to sign up: https://databricks.com/try-databricks. After you create your account, you will be provided with a couple of options to play with, as shown in Figure 1-6.

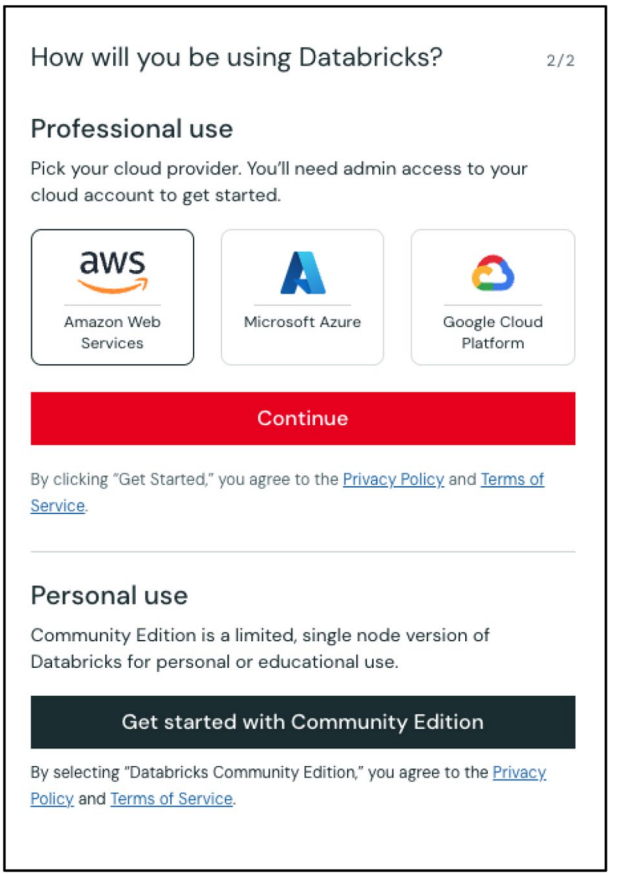

How will you be using Databricks?
2/2
Professional use
Pick your cloud provider. You'll need admin access to your cloud account to get started.
aws
Amazon Web Services
Microsoft Azure
Google Cloud Platform
Continue
By clicking "Get Started," you agree to the Privacy Policy and Terms of Service.
Personal use
Community Edition is a limited, single node version of Databricks for personal or educational use.
Get started with Community Edition
By selecting "Databricks Community Edition," you agree to the Privacy Policy and Terms of Service.

Figure 1-6. *Databricks options*

Select the Get Started with Community Edition button. The Community Edition is free. For most of the sections in this book, the Community Edition is sufficient. Alternatively, you can choose one of the cloud providers above and sign up for a 14-day free trial professional version. The Professional Edition provides additional capabilities over the Community Edition. However, you need an underlying cloud platform (AWS, Azure, or GCP) account.

As a word of caution—when you choose the 14-day free trial option, no cost will be incurred with your Databricks account (during the trial). However, you will incur costs from your cloud provider. Kindly ensure that cloud billing costs are monitored regularly to avoid surprises!

For the Community Edition, you should receive instructions in an email after you click Get Started. Follow the instructions to set up your account.

Create a Databricks Account

In summary:

- Create an account at https://databricks.com/try-databricks.
- Choose the Community Edition.
- Receive instructions in the email. Complete authentication and log in to Databricks Community Edition using the username and password you used to sign up.

Create a New Cluster

Figure 1-7 shows the Databricks landing page.

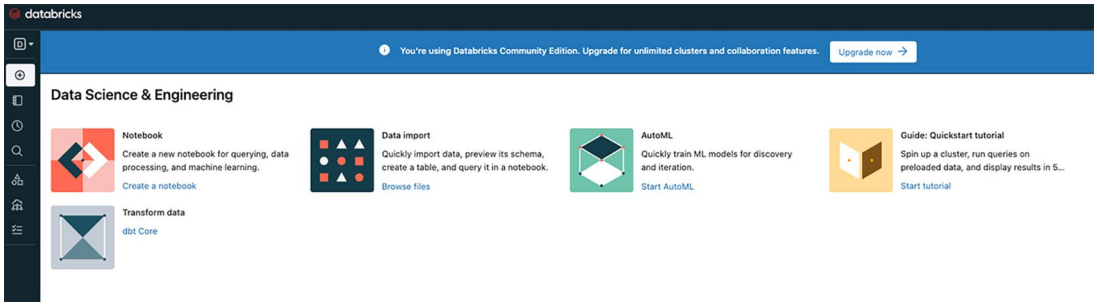

databricks
You're using Databricks Community Edition. Upgrade for unlimited clusters and collaboration features.
Upgrade now
Data Science & Engineering
Notebook
Create a new notebook for querying, data processing, and machine learning.
Create a notebook
Data import
Quickly import data, preview its schema, create a table, and query it in a notebook.
Browse files
AutoML
Quickly train ML models for discovery and iteration.
Start AutoML
Guide: Quickstart tutorial
Spin up a cluster, run queries on preloaded data, and display results in 5...
Start tutorial
Transform data
dbt Core

Figure 1-7. *Databricks landing page*

Navigate to Compute on the left pane and create a new compute instance. You must provide a cluster name (`pyspark_env`) and choose a Databricks runtime version (see Figure 1-8). The packages that we use for our data science demonstration are available in the ML runtime version. Go ahead and choose the latest runtime: 14.2 ML (Scala 2.12, Spark 3.5.0) version for this task and create the cluster.

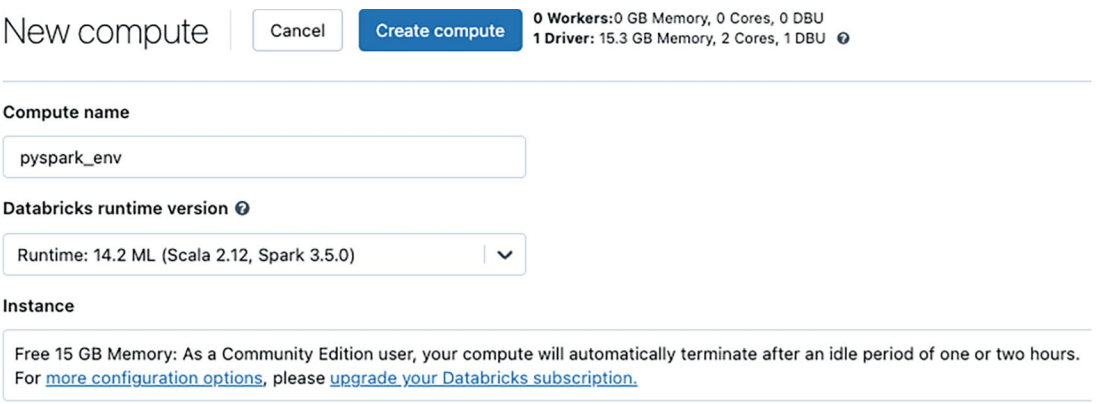

New compute
Cancel
Create compute
0 Workers:0 GB Memory, 0 Cores, 0 DBU
1 Driver: 15.3 GB Memory, 2 Cores, 1 DBU
Compute name
pyspark_env
Databricks runtime version
Runtime: 14.2 ML (Scala 2.12, Spark 3.5.0)
Instance
Free 15 GB Memory: As a Community Edition user, your compute will automatically terminate after an idle period of one or two hours.
For more configuration options, please upgrade your Databricks subscription.

Figure 1-8. *Cluster creation*

After you click Create Cluster, it takes time to spin up the cluster. Once the cluster is created, you'll see a green dot before the cluster name, as shown in Figure 1-9.

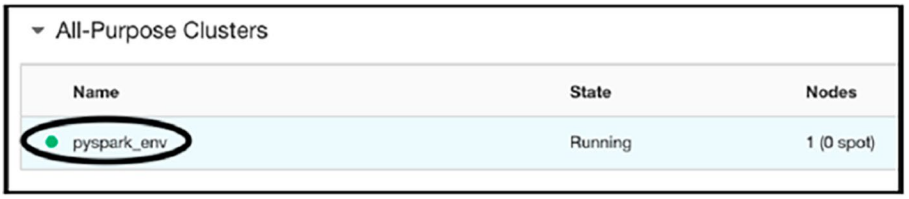


Figure 1-9. *Cluster creation is complete*

Create Notebooks

Click the Databricks icon on the top-left corner to navigate to the Welcome page (see Figure 1-7). Click Create a Notebook. You need to specify the cluster name you created in the previous task. This is shown in Figure 1-10.

Figure 1-10. *Databricks notebook*

To execute code in the cell, press Shift+Enter or use the Run Cell option in the right corner.

You are now all set up to run a PySpark notebook in Databricks. Easy, right? But you need to know one more thing before leaving this section. How do you import data into the Databricks environment? Let's look at that next.

Import Data Files into the Databricks Environment

In the left panel, click Create and select Table. The Create New Table screen opens, as shown in Figure 1-11.

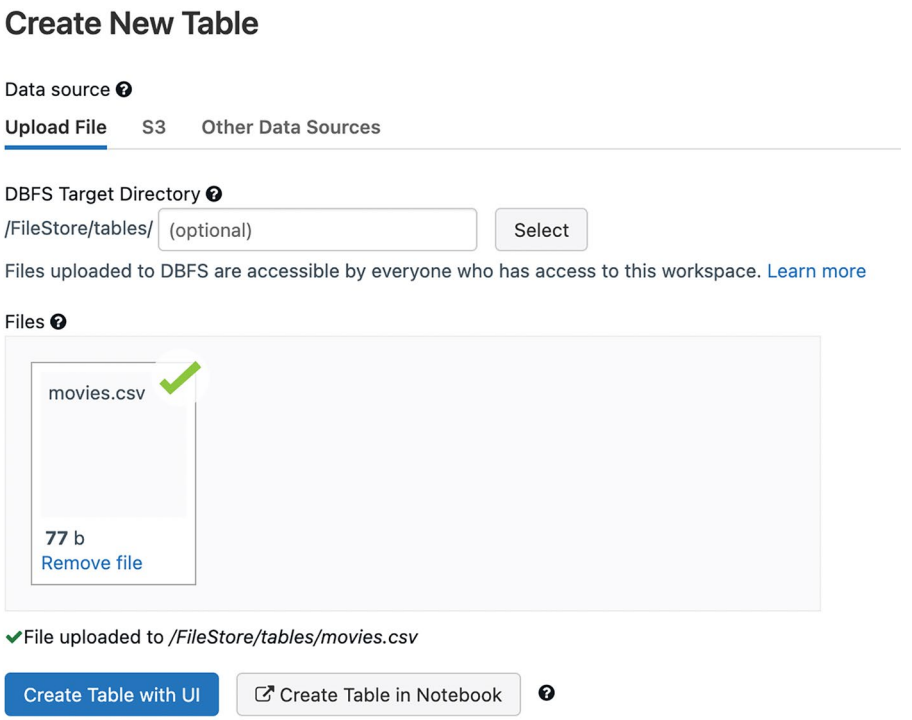


Figure 1-11. *Create new table*

The easiest option is the Upload File option, which we use most in this book. Create this simple CSV file and try to upload it:

```
movie, year
The Godfather,1972
The Shawshank Redemption, 1994
Pulp Fiction, 1994
```

You can now upload the file and choose the Create Table with UI option, then select the cluster you created before (`pyspark_env`). Click Preview Table to see the screen in Figure 1-12.

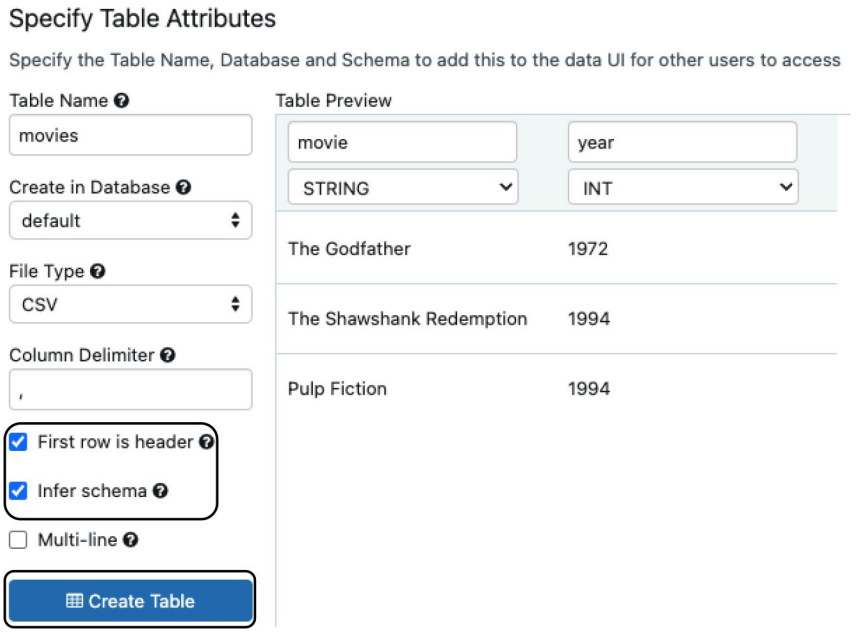


Figure 1-12. *Preview table*

Let's unpack the items in the Preview screen shown in Figure 1-12. Databricks will autodetect the delimiter of your file. However, if it fails to do this, you can specify your delimiter. Next, by checking the box, you explicitly mention that the first row is the header. When this option is not selected, the column names will be part of the table. Finally, ensure that the datatypes are appropriate before creating the table. Check the Infer Schema option to get columns autodetected with the correct datatypes. Optionally, you can also change the datatypes as per your requirements. Once all the table attributes are specified, click the Create Table button. You will notice a new table called `movie` in the `default` database.

Let's head back to the notebook using the Workspace option on the left panel and try to access this data. Usually, when you access the table using the PySpark SQL option, it creates a PySpark dataframe. We talk more about this later. For now, use the following code to access the data:

```
import pyspark
df = sqlContext.sql("select * from movies")
df.show()
```

You should get the output shown in Figure 1-13.

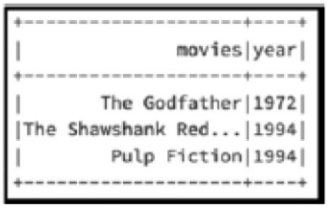


Figure 1-13. *Databricks PySpark output*

Pretty cool, right? You are all set to use the Databricks version.

GitHub Codespaces

Codespaces are ready-to-use cloud-based development environments offered by GitHub. GitHub code can be executed on the browser-based Visual Studio Code (VS Code) with a few clicks. Personal GitHub accounts are eligible for free use of codespaces, but there are usage quota limitations. Refer to this link for additional details: https://docs.github.com/en/codespaces/overview.

Sign up for a GitHub account (if you don't have one) and log in. Navigate to the GitHub repository associated with this book: https://github.com/Apress/applied-data-science-using-pyspark.

Select Create Codespace on Main to launch VS Code on the browser (see Figure 1-14).

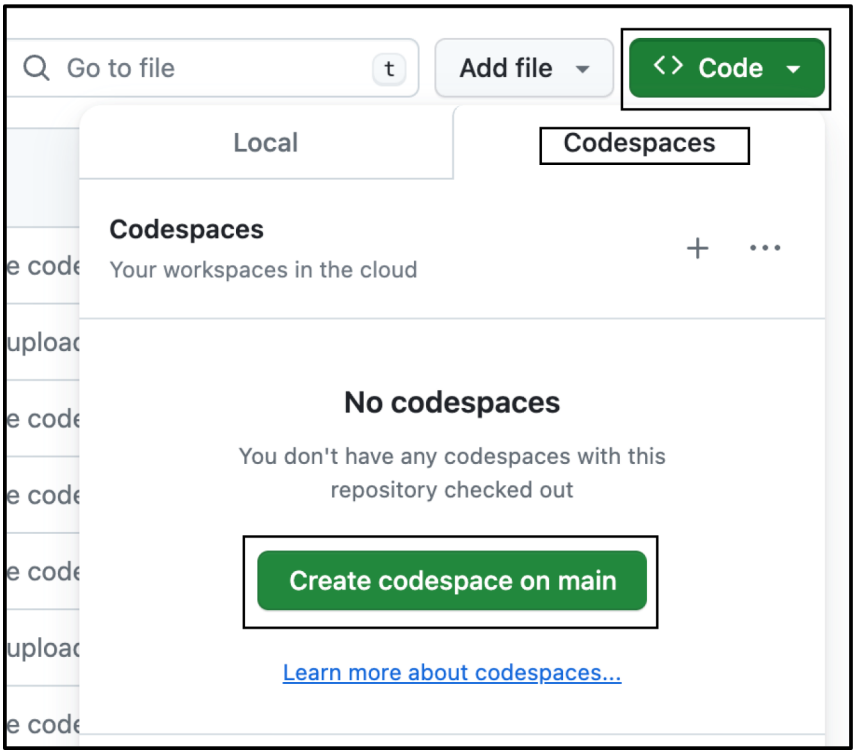


Figure 1-14. *GitHub codespaces*

The VS Code landing page is shown in Figure 1-15. Navigate to Settings (bottom left) and install the Python and Jupyter extensions offered by Microsoft (see Figure 1-16).

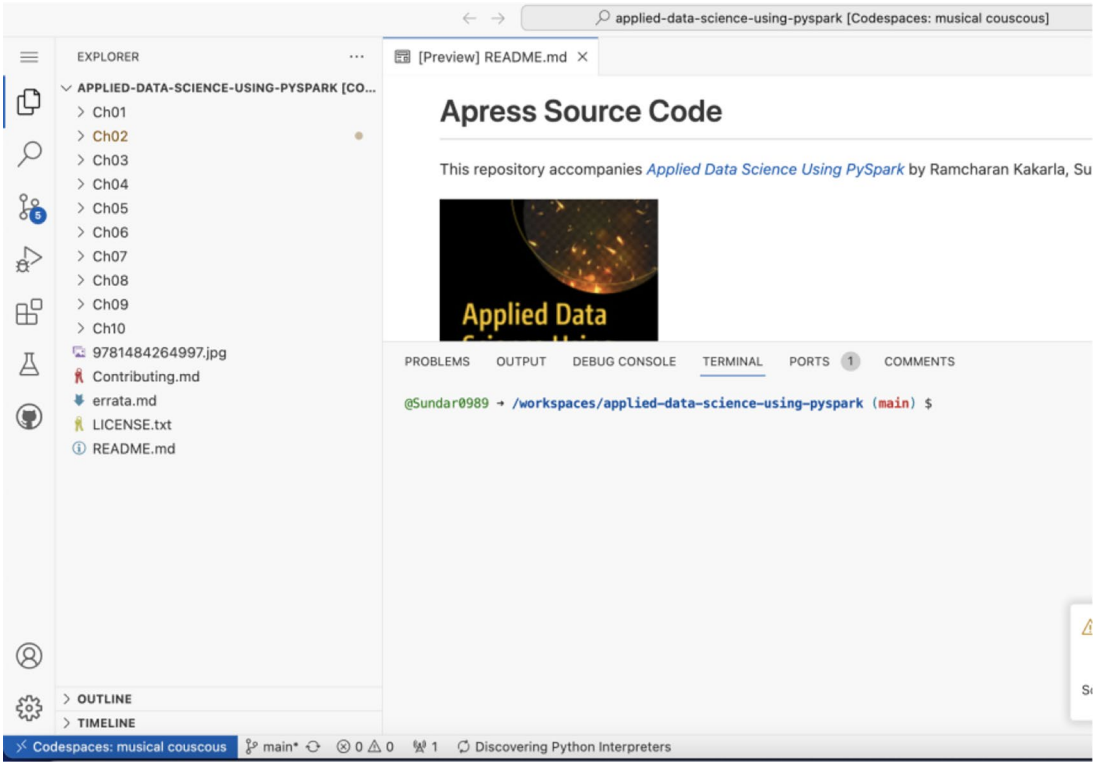


Figure 1-15. *VS Code landing page (browser)*

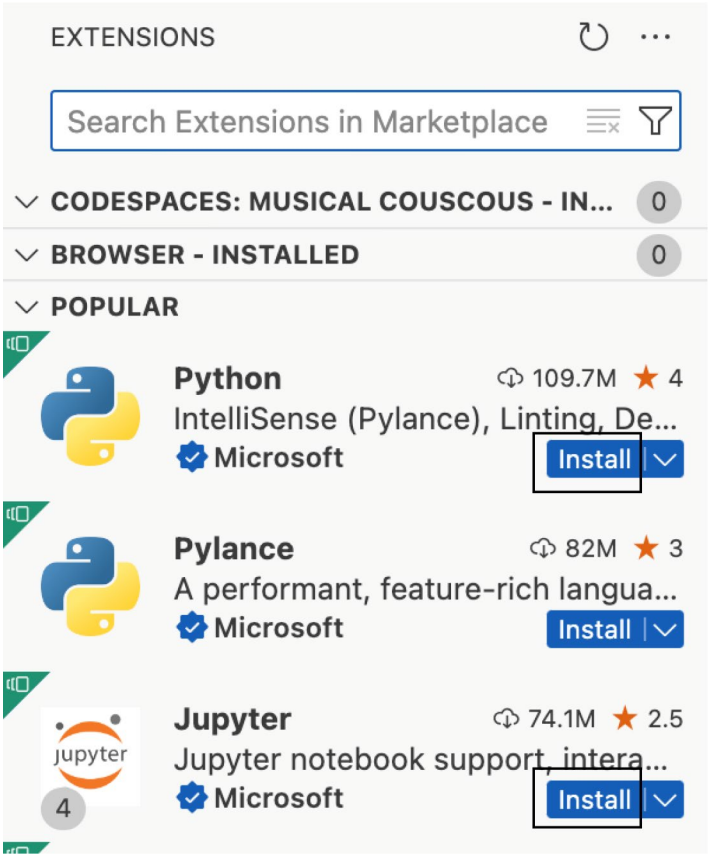


Figure 1-16. *The Microsoft Python and Jupyter extensions*

In the terminal, run `pip install pyspark`, as shown in Figure 1-17. This should install the latest Spark 3.5.0 version.

```
@Sundar0989 → /workspaces/applied-data-science-using-pyspark (main) $ pip install pyspark
Collecting pyspark
  Downloading pyspark-3.5.0.tar.gz (316.9 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 316.9/316.9 MB 5.6 MB/s eta 0:00:00
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Collecting py4j==0.10.9.7 (from pyspark)
  Downloading py4j-0.10.9.7-py2.py3-none-any.whl (200 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 200.5/200.5 kB 6.5 MB/s eta 0:00:00
Building wheels for collected packages: pyspark
  Building wheel for pyspark (pyproject.toml) ... done
  Created wheel for pyspark: filename=pyspark-3.5.0-py2.py3-none-any.whl size=317425345 sha256=b1a8fe01212d45256874c8
  Stored in directory: /home/codespace/.cache/pip/wheels/41/4e/10/c2cf2467f71c678cfc8a6b9ac9241e5e44a01940da8fbb17fc
Successfully built pyspark
Installing collected packages: py4j, pyspark
Successfully installed py4j-0.10.9.7 pyspark-3.5.0
```

Figure 1-17. *The VS Code terminal*

Now you can test the code execution in VS Code. Open the `Ch02/ Chapter2_Notebook` folder to launch the `Spark_demonstration.ipynb` notebook. Click Select Kernel (in the top-right corner). A pop-up window should appear in the middle of the screen. Select Python Environment and Recommended Python version, as shown in Figure 1-18.

Figure 1-18. *The Python Environment*

In this case, the Python version is 3.10.3 and the Spark version is 3.5.0. Depending on when you run the notebooks, your Python and Spark versions may vary slightly.

You are now all set to execute the code. Run the first two cells to ensure the required packages are installed. You should see results like what's shown in Figure 1-19.

```
# Print PySpark and Python versions
import sys
print('Python version: '+sys.version)
print('Spark version: '+spark.version)
```

✓ 0.0s

```
Python version: 3.10.13 (main, Jan 10 2024, 19:45:45) [GCC 9.4.0]
Spark version: 3.5.0
```

Figure 1-19. *Notebook package verification*

From here, minor tweaks may be required to execute the code successfully. For example, in cell 3, update `file_location` to point to the right file location as shown here (see Figure 1-20):

```
file_location = "../Chapter2_Data/movie_data_part1.csv"
```

```
# Read data
# old file_location = "movie_data_part1.csv"

file_location = "../Chapter2_Data/movie_data_part1.csv"
file_type = "csv"
infer_schema = "false"
first_row_is_header = "true"
delimiter = "|"

df = spark.read.format(file_type)\
.option("inferSchema", infer_schema)\
.option("header", first_row_is_header)\
.option("sep", delimiter)\
.load(file_location)
```

Figure 1-20. *Notebook new file_location*

Hooray, you're all set to execute PySpark code with GitHub codespaces. This eliminates the need for complex local environment setup and maintenance.

Basic Operations

Congratulations on setting up your PySpark environment. In this section, you use some toy codes to play with the environment you just created. All the demonstrations shown hereafter should work irrespective of the setup option you chose.

Upload Data

This section is intended for users who chose either local or Docker-based installation. Databricks users should upload their files separately and reference them in the notebook. Codespace users can upload the file to GitHub and use it in their code.

Once you invoke the Jupyter Notebook, you can upload data directly to your environment (see Figure 1-21). Now create a simple CSV file and name it `movies.csv`.

```
movie, year
The Godfather,1972
The Shawshank Redemption, 1994
Pulp Fiction, 1994
```

Figure 1-21. *Upload files*

Access Data

Once the file is uploaded, jump to your PySpark notebook and run the following code:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
df = spark.read.csv("movies.csv", header=True, sep=',')
df.show()
```

Calculate Pi

This is a typical example that you see online or in textbooks. It will calculate the value of pi using 100 million samples to test the PySpark operation. Once the operation is complete, it will provide you with the value of Pi. Use the following code:

```
# Importing the SparkContext module from PySpark
from pyspark import SparkContext
# Creating a SparkContext object
sc = SparkContext.getOrCreate()
# Importing the random module
import random
NUM_SAMPLES = 100000000 # Define the samples
# Function to check if a point lies inside
def inside(p):
    x, y = random.random(), random.random()
    return x*x + y*y < 1
# parallelize the computation
```

```
count = sc.parallelize(range(0, NUM_SAMPLES)).filter(inside).count()
# calculate the estimate value of pi
pi = 4 * count / NUM_SAMPLES
print("Pi is roughly", pi)
```

Summary

- You created your own PySpark instance with one of the following: local installation using Anaconda, Docker-based installation, Databricks Community Edition, or GitHub codespaces.
- You know the steps to access local files in the Jupyter Notebook environment.
- You created a PySpark dataframe using a `movies.csv` file and printed the output.
- You ran a small PySpark command to calculate the value of Pi.

Great job! In the next chapter, we delve more into PySpark operations and take you through some interesting topics, like feature engineering and visualization. Keep learning and stay tuned.

CHAPTER 2

PySpark Basics

This chapter will help you understand the basic operations of PySpark. You are encouraged to set up the PySpark environment and try the following operations on any dataset you choose for enhanced understanding. Given the breadth of Spark, we will cover basic concepts before delving into data-wrangling tasks. This chapter illustrates the most frequent data operations in PySpark that you might encounter on a daily basis.

This chapter covers the following topics:

- Background of PySpark
- PySpark Resilient Distributed Dataset (RDD) and DataFrames
- Data manipulations

PySpark Background

Before leaping to PySpark, you should learn about what Spark is and its advantages. Spark is an engine for processing large amounts of data. It is written in Scala and runs on a Java virtual machine. A detailed representation of the Apache Spark ecosystem is presented in Figure 2-1.

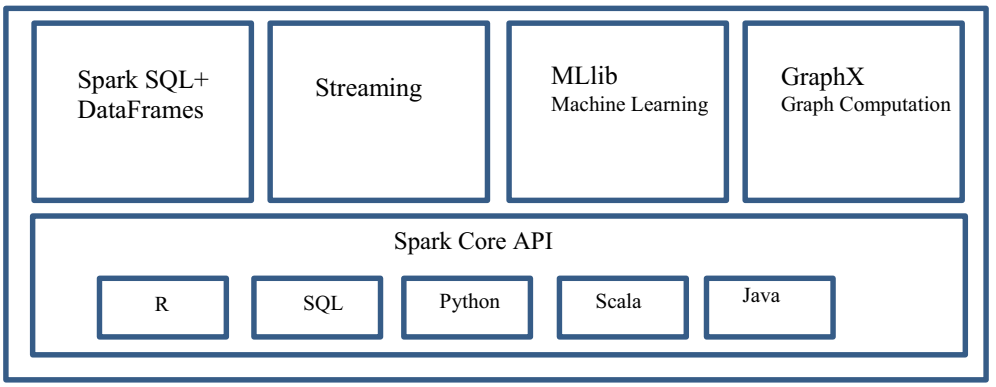


Figure 2-1. *Overview of the Spark ecosystem*

© Ramcharan Kakarla, Sundar Krishnan, Balaji Dhamodharan and Venkata Gunnu 2024
R. Kakarla et al., *Applied Data Science Using PySpark*, https://doi.org/10.1007/979-8-8688-0820-3_2

The Spark core is the underlying general execution engine for the platform. Spark has built-in components for SQL, streaming, machine learning, and GraphX (graph processing tasks).

- **Spark SQL** is best suited for working with structured data. It enables you to run hive queries at blazing speeds, up to 100x faster with the existing data.
- **Streaming** supports nontraditional real-time datasets. It enables interactivity and analytical operations on streaming data and integrates well with popular data sources, including HDFS, Flume, Kafka, and Twitter.
- **Machine learning** is a scalable library built on Spark that enables you to run different algorithms for specific use cases. It has most of the widely used algorithms off the shelf.
- **GraphX** is a graph computation engine built on top of Spark that enables you to interactively create, transform, and reason about graph-structured data at scale.

Speed is synonymous with Spark and is one of the key drivers of its recent popularity. At its core, Spark achieves this speed via parallel processing and in-memory computations. How does Spark achieve this parallel processing? Well, the answer is Scala, Spark's native language. Scala is functional-based, which fits well with parallel processing. Its ease of use for data transformations on large datasets makes it a preferred tool. Spark also has a rich set of libraries, including machine learning, SQL, and streaming support. It is one of the most versatile tools for taming big data in a unified environment.

Let's step back and see how Spark runs on a cluster to achieve parallel processing. Spark derives its basics from MapReduce. MapReduce is a popular framework for processing large datasets. It consists of two major stages. The map stage is where all the tasks are split among different computers. The reduce stage is where all the keys are shuffled logically and further reduced for calculating meaningful aggregations or doing data transformations. Figure 2-2 illustrates how Spark works in cluster mode.

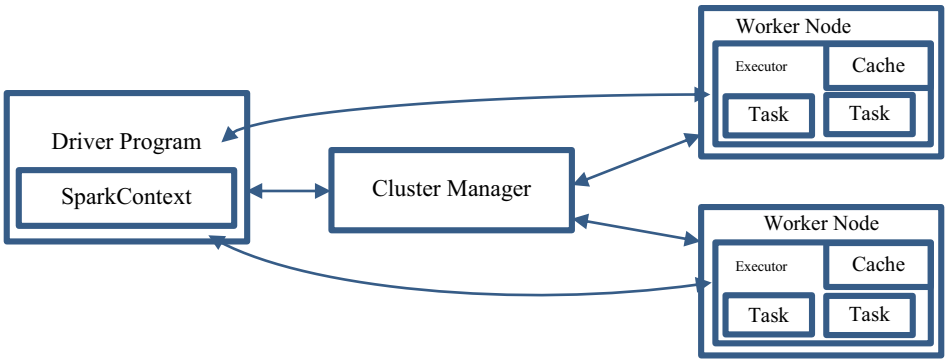


Figure 2-2. *Overview of Spark cluster mode*

When Spark is initiated, an independent process is spun up and coordinated by SparkContext or Spark Session, as indicated on the left side of Figure 2-2. The Cluster Manager assigns the work to the workers. Two workers are shown in Figure 2-2. In practice, there could be *n* number of workers. The task executes a unit of work to the dataset and outputs a new partition dataset. As multiple operations are executed to the data, they benefit from the in-memory computation powered by caching data across these operations. Depending on the type of operation, aggregation results are either sent back to the driver or can be saved onto the disk.

The Directed Acyclic Graph (DAG) serves as a foundational concept within Spark, crucial for both representation and optimization of operational flows. When code is submitted to the Spark console, it generates an operator graph, which is subsequently forwarded to the DAG scheduler upon triggering an action, such as aggregation or data saving. The scheduler then divides the operator graphs into stages. Every step can contain several data partitions, and the DAG scheduler bundles all these individual operator graphs together. The task of workers is to execute these tasks on the executors.

Let's review some concepts and properties of Spark before hopping to RDDs and DataFrames. You may come across these terms in different sections of the book:

> **Immutable:** Something immutable can be considered a data unit that cannot be reversed after a transformation. This is extremely important for achieving consistency in computations, especially in a parallel computation setting.

Fault-Tolerant: In the case of worker failure during data transformation, the data can be recovered by replaying the transformation on that partition in lineage to achieve the same computation. This eliminates data replication across multiple nodes.

Lazy Evaluation: All transformations in Spark are lazy, meaning computations are not done right away. Instead, Spark remembers all the transformation logic and they are computed only when an action is called. An action could be either returning a result to the driver or saving a dataset.

Type Safety: This indicates that the compiler knows the expected columns and their datatypes.

Data Serialization: This is a method of translating data structures to a specific format to effectively manage the source and destination. By default, Apache Spark uses Java serialization.

Data Locality: Processing tasks are optimized by placing the execution code close to the processed data.

Predicate Pushdown: This technique allows Spark to process only the required data by using filter conditions. Only required files and partitions are accessed, thus reducing the disk I/O.

Accumulators: These are global variables assigned to the executors that can be added through the associative and commutative properties.

Catalyst Optimizer: This is an extensible optimizer used to implement Spark SQL. it supports rule-based and costbased optimization.

So now you may be wondering where PySpark sits in this ecosystem. PySpark can be conceived as a Python-based wrapper on top of the Scala API. PySpark communicates with Spark's Scala-based API through the Py4J library. This library allows Python to interact with the Java virtual machine code. In the next section, you look at the ways in which you can interact with data in PySpark via RDDs and DataFrames. We focus more on the DataFrames, and you will gain deeper insight into why we made that decision in the next section.

PySpark Resilient Distributed Datasets (RDDs) and DataFrames

An *RDD* is an immutable distributed collection of elements of your data. It is partitioned across nodes in your cluster, enabling interaction with a low-level API for parallel processing.

RDDs are flexible in handling structured, semi-structured, and unstructured data. They are best suited for semi- and unstructured data. They can be used to access these data elements with ease and control for any low-level transformation. RDDs are part of Spark's core libraries and only need the Spark context for their operations.

RDDs do come with their fair share of challenges. Optimizations and performance gains available with DataFrames are absent with RDDs. If you are working with structured data, your go-to should always be DataFrames. You can always change an RDD to a DataFrame as long as you can define a schema.

A *dataset* is a distributed collection of data. It has benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL's optimized execution engine. A dataset can be constructed from JVM objects and then manipulated using functional transformations (`map`, `flatMap`, `filter`, etc.). The Dataset API is available only in Scala and Java. Python does not have support for the Dataset API. But due to Python's dynamic nature, many of the benefits of the Dataset API are already available (i.e., you can access the field of a row by name naturally: `row.columnName`). The case for R is similar.

A *DataFrame* is a dataset ordered into columns. It holds data in column and row format, like a table. You can perceive the columns to be the variables and the rows the associated data points. It is conceptually equivalent to a table in a relational database or a dataframe in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources, such as structured data files, tables in Hive, external databases, or existing RDDs. The DataFrame API is available in Scala, Java, Python, and R. SQLContext is needed to program DataFrames, as they lie in the SparkSQL area of the Spark ecosystem. You can always convert a DataFrame to an RDD if needed.

Table 2-1 captures the differences between RDDs, DataFrames, and datasets in a nutshell. Remember: We use DataFrames as the primary way to interact with data in this book.

Table 2-1. *Features by RDD, DataFrame, and Dataset in Spark*

Feature	RDD	DataFrame	Dataset
Immutable	Yes	Yes	Yes
Fault-Tolerant	Yes	Yes	Yes
Type-Safe	Yes	No	Yes
Schema	No	Yes	Yes
Optimizer Engine	NA	Catalyst	Catalyst
Execution Optimization	No	Yes	Yes
API Level Manipulation	Low Level	High Level	High Level
Language Support	Java, Scala, Python, R	Java, Scala, Python, R	Java, Scala

The Spark SQL `DataType` class is the base class of all datatypes, and they are primarily used in all DataFrames (see Table 2-2).

Table 2-2. *Datatypes*

Datatype	Broad Classification
`BooleanType`	Atomic type: It is an internal type used to represent everything that is not null, arrays, structs, or maps.
`BinaryType`	
`DateType`	
`StringType`	
`TimestampType`	

(*continued*)

Table 2-2. (*continued*)

Datatype	Broad Classification
ArrayType	Non-atomic types
MapType	
StructType	
CalendarIntervalType	
NumericType	
ShortType	
IntegerType	
LongType	
FloatType	
DoubleType	
DecimalType	
ByteType	
HiveStringType	
ObjectType	
NullType	

In the previous chapter, we introduced different ways you can install and use Spark. Now, let's get to the interesting part and see how you can perform data operations. For the purpose of this book, we use the Docker version of PySpark, running on a single machine. If you have a version of PySpark installed on a distributed system, we encourage using it to unleash the power of parallel computing. There is no difference in

programming or commands between running your operations on a single standalone machine and doing so on a cluster. Keep in mind that you will lose out on the processing speeds if you are using a single machine.

To demonstrate data operations, we use a dataset from *themoviedatabase* (TMDB: https://www.themoviedb.org/). TMDB is a community-built movie and TV database. We used an API to extract a sample of their data. This data contains information on 44,000 movies released internationally. We have the information from Table 2-3 in the dataset. This dataset is in the form of a csv file, which is pipe delimited. This dataset will be shared in the chapter's GitHub page.

Table 2-3. *Metadata*

Variable	Description
belongs_to_collection	Indicates whether the movie belongs to a collection; collection is specified if it exists.
budget	Movie's budget.
id	Unique identifier for the movie.
original_language	Original language in which movie was produced.
original_title	Title of the movie.
overview	Summary of the movie.
popularity	Popularity index of the movie.
production_companies	List of companies that produced the movie.
production_countries	Country where the movie was produced.
release_date	Movie release date.
revenue	Movie collection, missing is represented by 0.
runtime	Movie runtime in minutes.
status	Indicates whether movie was released or not.
tagline	Movie tagline.
title	Movie alias English title.
vote_average	Average vote rating by viewers.

Data Manipulations

We work with this data through the end of this chapter as we demonstrate data manipulations.

Once you have set up your connection to Spark using any of the methods demonstrated in the previous chapter, initiate the PySpark session, which can be done using the following code:

```
from pyspark.sql import SparkSession
spark=SparkSession.builder.appName("Data_Wrangling").getOrCreate()
```

Note The preceding step is universal for any PySpark program.

As you'll recall from Figure 2-1, `SparkSession` is the entry point and connects your PySpark code to the Spark cluster. By default, all the configured nodes used for the execution of your code are in a cluster mode setting.

Now, let's read your first dataset. We are using a dataset located on our local machine in the path that we exposed to the Docker session. We are using the Docker method for demonstration purposes. Spark runs on Java 8/11, Scala 2.12, Python 2.7+/3.4+, and R 3.1+. With Java 8 prior to version 8u92, support is deprecated as of Spark 3.0.0. For Python 2 and Python 3 prior to version 3.6, support is deprecated as of Spark 3.0.0. For R prior to version 3.4, support is deprecated as of Spark 3.0.0. Make sure you have the right Java versions for Spark compatibility if you are using a local version. As with Python, all PySpark DataFrame variables are case sensitive.

Reading Data from a File

```
# This is the location where the data file resides
file_location = "movie_data_part1.csv"
# Type of file, PySpark also can read other formats such as json,
parquet, orc
file_type = "csv"
# As the name suggests, it can read the underlying existing schema
if exists
infer_schema = "False"
```

```
#You can toggle this option to True or False depending on whether you have
header in your file or not
first_row_is_header = "True"
# This is the delimiter that is in your data file
delimiter = "|"
# Bringing all the options together to read the csv file
df = spark.read.format(file_type) \
.option("inferSchema", infer_schema) \
.option("header", first_row_is_header) \
.option("sep", delimiter) \
.load(file_location)
```

If your dataset is not in a flat file format and exists as a hive table, you can read it easily via a single command, as shown next.

Reading Data from a Hive Table

Use this code to read data from a hive table:

```
df = spark.sql("select * from database.table_name")
```

Note Spark only maps the relationship upon reading a data source and will not bring any data into memory.

Executing the preceding commands will make your data accessible to Spark, and you will be all set to experience the magic of PySpark.

Once developers have successfully read the data from the source, one of the first questions they have is often, how can I know the metadata information? One of the handy ways to read metadata is through a simple command.

Reading Metadata

The output is presented in Figure 2-3. You can quickly identify the variables with their datatypes using this function:

```
df.printSchema()
```

```
root
 |-- belongs_to_collection: string (nullable = true)
 |-- budget: string (nullable = true)
 |-- id: string (nullable = true)
 |-- original_language: string (nullable = true)
 |-- original_title: string (nullable = true)
 |-- overview: string (nullable = true)
 |-- popularity: string (nullable = true)
 |-- production_companies: string (nullable = true)
 |-- production_countries: string (nullable = true)
 |-- release_date: string (nullable = true)
 |-- revenue: string (nullable = true)
 |-- runtime: string (nullable = true)
 |-- status: string (nullable = true)
 |-- tagline: string (nullable = true)
 |-- title: string (nullable = true)
 |-- vote_average: string (nullable = true)
```

Figure 2-3. *Ouput of printSchema*

There are multiple ways to achieve the preceding output. The same information can also be extracted by using the following command in a Python list format, which is more desirable for further data operations:

```
df.dtypes
```

You can also use `df.columns` to list the columns without the datatype information.

Counting Records

One other quick check you can perform on a dataset is to observe the total number of records. This can be done via the following command:

```
df.count()
```

Output:

```
43998
```

You can wrap this in a `print` function in Python to get a more descriptive output, as follows:

```
print('The total number of records in the movie dataset is '+str(df.
count()))
```

Output:

```
The total number of records in the movie dataset is 43998
```

We know from previous experience with this dataset that a few of the columns are dictionaries and are not useful in their raw form. Some of the variables contain descriptive information. Let's select a few random variables and see how they look.

Subset Columns and View a Glimpse of the Data

Again, there are multiple ways to achieve the same output (see Figure 2-4) of the following:

```
# Defining a list to subset the required columns
select_columns=['id', 'budget', 'popularity', 'release_date', 'revenue',
'title']
# Subsetting the required columns from the DataFrame
df=df.select(*select_columns)
# The following command displays the data; by default it shows top 20 rows
df.show()
```

```
+-----+-------+------------------+------------+-------+--------------------+
|   id| budget|        popularity|release_date|revenue|               title|
+-----+-------+------------------+------------+-------+--------------------+
|43000|      0|             2.503|  1962-05-23|      0|The Elusive Corporal|
|43001|      0|              5.51|  1962-11-12|      0|  Sundays and Cybele|
|43002|      0|              5.62|  1962-05-24|      0|Lonely Are the Brave|
|43003|      0|             7.159|  1975-03-12|      0|          F for Fake|
|43004| 500000|             3.988|  1962-10-09|      0|Long Day's Journe...|
|43006|      0|             3.194|  1962-03-09|      0|          My Geisha|
|43007|      0|             2.689|  1962-10-31|      0|Period of Adjustment|
|43008|      0|             6.537|  1959-03-13|      0|    The Hanging Tree|
|43010|      0|             4.297|  1962-01-01|      0|Sherlock Holmes a...|
|43011|      0|             4.417|  1962-01-01|      0|  Sodom and Gomorrah|
|43012|7000000|4.7219999999999995|  1962-11-21|4000000|         Taras Bulba|
|43013|      0|             2.543|  1962-04-17|      0|The Counterfeit T...|
|43014|      0|             4.303|  1962-10-24|      0|     Tower of London|
|43015|      0|             3.493|  1962-12-07|      0|Varan the Unbelie...|
|43016|      0|             2.851|  1962-01-01|      0|Waltz of the Tore...|
|43017|      0|             4.047|  1961-10-11|      0|         Back Street|
|43018|      0|             2.661|  1961-06-02|      0|Gidget Goes Hawaiian|
|43019|      0|             3.225|  2010-05-28|      0|Schuks Tshabalala...|
|43020|      0|              5.72|  1961-06-15|      0|The Colossus of R...|
|43021|      0|             3.292|  2008-08-22|      0|          Sex Galaxy|
+-----+-------+------------------+------------+-------+--------------------+
only showing top 20 rows
```

Figure 2-4. *Output of df.show()*

All the operations are done in a single statement, as follows:

```
df.select('id', 'budget', 'popularity', 'release_date', 'revenue',
'title').show()
```

You also have the option of selecting the columns by index instead of selecting the names from the original DataFrame:

```
df.select(df[2],df[1],df[6],df[9],df[10],df[14]).show()
```

Be cautious when using the preceding option if you already used a first method to subset. It may throw an `index out of range` error. This is because the new DataFrame will contain only a subset of the columns. You can also change the number of rows displayed in the `show` option to a required number by using the following statement. (In this example, we set the number to 25, but it can be changed to any *n*.)

```
df.show(25,False)
```

Note Using `False` not only allows you to override the default setting of 20 rows, but it also displays the truncated content, if any.

Missing Values

You can calculate the missing values in a single column or in multiple columns by using the built-in functions in PySpark, as follows:

```
from pyspark.sql.functions import *
df.filter((df['popularity']=='')|df['popularity'].isNull()|isnan(df
['popularity'])).count()
```

Output:

```
215
```

Let's break down the preceding command and see what is happening. First, the `filter` function is used on the DataFrame and passes multiple conditions based on an `OR` condition, which is represented by `|` in the expression. In the first condition, we search for a null string present in the `popularity` column. In the second condition, we use the

`.isNull()` operator, which returns true when popularity is null; null represents no value or nothing. In the third condition, we use `isnan` to identify NaN (not a number). These are usually a result of mathematical operations that are not defined.

If you need to calculate all the missing values in the DataFrame, you can use the following command:

```
df.select([count(when((col(c)=='') | col(c).isNull() |isnan(c), c)).alias(c)
for c in df.columns]).show()
```

This command selects all the columns and runs the preceding missing checks in a loop. Then `when` condition is used here to subset the rows that meet the missing value criteria (see Figure 2-5).

```
+---+------+----------+------------+-------+-----+
| id|budget|popularity|release_date|revenue|title|
+---+------+----------+------------+-------+-----+
|125|   125|       215|         221|    215|  304|
+---+------+----------+------------+-------+-----+
```

Figure 2-5. *Ouput of missing values across all columns of a DataFrame*

Note Make sure to import `pyspark.sql.functions` before you run any sort operations.

One-Way Frequencies

Okay, let's see how you can calculate the frequencies of categorical variables (see Figure 2-6). Terminology alert: Categorical variables are string variables that exist in a dataset. Let's verify if there are any repetitive titles in the dataset, using the following code:

```
df.groupBy(df['title']).count().show()
```

```
+--------------------+-----+
|               title|count|
+--------------------+-----+
|   The Corn Is Green|    1|
|Meet The Browns -...|    1|
|Morenita, El Esca...|    1|
| Father Takes a Wife|    1|
|The Werewolf of W...|    1|
|My Wife Is a Gang...|    1|
|Depeche Mode: Tou...|    1|
|  A Woman Is a Woman|    1|
|History Is Made a...|    1|
|      Colombian Love|    1|
|        Ace Attorney|    1|
|     Not Like Others|    1|
|40 Guns to Apache...|    1|
|          Middle Men|    1|
|         It's a Gift|    1|
|    La Vie de Bohème|    1|
|Rasputin: The Mad...|    1|
|The Ballad of Jac...|    1|
|         How to Deal|    1|
|             Freaked|    1|
+--------------------+-----+
only showing top 20 rows
```

Figure 2-6. *Output of one-way frequency*

The preceding command gives you the number of times a title has appeared in a dataset. Often, you might like to see sorted data, so you could use the following (see Figure 2-7):

```
df.groupby(df['title']).count().sort(desc("count")).show(10, False)
```

```
+--------------------+-----+
|title               |count|
+--------------------+-----+
|null                |304  |
|Les Misérables      |8    |
|The Three Musketeers|8    |
|Cinderella          |8    |
|The Island          |7    |
|A Christmas Carol   |7    |
|Hamlet              |7    |
|Dracula             |7    |
|Frankenstein        |7    |
|The Lost World      |6    |
+--------------------+-----+
only showing top 10 rows
```

Figure 2-7. *Output of one-way frequency sorted*

Note groupby is an alias for groupBy; both variants work.

There are quite a few titles that are repeated. It looks like having a missing title is the most common issue. Real-world data can be messy. This is good, but you may want to fine-tune the results by eliminating any missing values and adding an additional filter to limit it only to titles that occurred more than four times.

Sorting and Filtering One-Way Frequencies

Let's first filter the values that are not null. We used the not-equals sign (! = and ~) to create a temporary dataset, which is done intentionally to demonstrate the use of the not condition.

```
# Subsetting and creating a temporary DataFrame to eliminate any
missing values
df_temp=df.filter((df['title']!=")&(df['title'].isNotNull()) &
(~isnan(df['title'])))
# Group by 'title', count occurrences, filter titles occurring more than
four times, sort in descending order, and display top results
df_temp.groupby(df_temp['title']).count().filter("`count` >4").
sort(col("count").desc()).show(10,False)
```

If you don't specify the desc function, the results will be sorted in ascending order by default (see Figure 2-8).

```
+--------------------+-----+
|title               |count|
+--------------------+-----+
|Les Misérables      |8    |
|The Three Musketeers|8    |
|Cinderella          |8    |
|Hamlet              |7    |
|The Island          |7    |
|Dracula             |7    |
|A Christmas Carol   |7    |
|Frankenstein        |7    |
|Framed              |6    |
|The Lost World      |6    |
+--------------------+-----+
only showing top 10 rows
```

Figure 2-8. *Output of filtered and sorted version of one-way frequency*

```
# The following command is to find the number of titles that are repeated
four times or more
df_temp.groupby(df_temp['title']).count().filter("`count` >=4").
sort(col("count").desc()).count()
```

Output:

```
43
# The following command is to delete any temporary DataFrames that we
created in the process
del df_temp
```

In the preceding commands, you may have observed that there are multiple ways to achieve the same result. As you become fluent in the language, you may develop your own preferences.

Casting Variables

As you have observed from the metadata, all the variables are indicated as strings. After observing the data, you know some of these fields are integers, floats, or date objects. Some of the operations can lead to misleading results if the datatypes are not correctly identified. We strongly recommend you do the due diligence of identifying the right datatype for any of your analyses. You will now see how to cast some of these variables to the right datatypes.

If you are not careful in identifying the datatypes, you may experience data loss on casting. For example, if you are converting a string column to a numeric column, the resulting column can be all nulls. Try this with the `tagline` column in the demo dataset and observe the result. It is good practice to explore the dataset before applying any transformations. Some typos in the manual data (e.g., 12,1 instead of 12.1) can also result in unexpected results.

First print the datatypes before casting, convert the `budget` variable to a `float`, and observe the change (see Figures 2-9 and 2-10).

```
#Before Casting
df.dtypes
```

```
[('id', 'string'),
 ('budget', 'string'),
 ('popularity', 'string'),
 ('release_date', 'string'),
 ('revenue', 'string'),
 ('title', 'string')]
```

Figure 2-9. *Datatypes before casting*

```
#Casting
df = df.withColumn('budget',df['budget'].cast("float"))
#After Casting
df.dtypes
```

```
[('id', 'string'),
 ('budget', 'float'),
 ('popularity', 'string'),
 ('release_date', 'string'),
 ('revenue', 'string'),
 ('title', 'string')]
```

Figure 2-10. *Datatypes after casting*

The `cast` function is used. Also note that we used the additional function called `.withColumn`. This is one of the most common functions used in PySpark. It is used to update the values of datatypes, as well as rename and convert them. It can also create new columns (see Figures 2-11 and 2-12). You will have a better appreciation of its use further down the line. You may be wondering how you can extend the function to multiple columns. If you already know your datatypes, you can use the lists of identified variables and iterate them over a loop.

```
#Importing necessary libraries
from pyspark.sql.types import *
#Identifying and assigning lists of variables
int_vars=['id']
float_vars=['budget', 'popularity', 'revenue']
date_vars=['release_date']
#Converting integer variables
for column in int_vars:
     df=df.withColumn(column,df[column].cast(IntegerType()))
```

```
# Converting float variables
for column in float_vars:
     df=df.withColumn(column,df[column].cast(FloatType()))
# Converting float variables
for column in date_vars:
     df=df.withColumn(column,df[column].cast(DateType()))
df.dtypes
```

```
[('id', 'int'),
 ('budget', 'float'),
 ('popularity', 'float'),
 ('release_date', 'date'),
 ('revenue', 'float'),
 ('title', 'string')]
```

Figure 2-11. *Datatypes of all variables after casting*

```
df.show(10,False)
```

```
+-----+--------+----------+------------+-------+---------------------------------------+
|id   |budget  |popularity|release_date|revenue|title                                  |
+-----+--------+----------+------------+-------+---------------------------------------+
|43000|0.0     |2.503     |1962-05-23  |0.0    |The Elusive Corporal                   |
|43001|0.0     |5.51      |1962-11-12  |0.0    |Sundays and Cybele                     |
|43002|0.0     |5.62      |1962-05-24  |0.0    |Lonely Are the Brave                   |
|43003|0.0     |7.159     |1975-03-12  |0.0    |F for Fake                             |
|43004|500000.0|3.988     |1962-10-09  |0.0    |Long Day's Journey Into Night          |
|43006|0.0     |3.194     |1962-03-09  |0.0    |My Geisha                              |
|43007|0.0     |2.689     |1962-10-31  |0.0    |Period of Adjustment                   |
|43008|0.0     |6.537     |1959-03-13  |0.0    |The Hanging Tree                       |
|43010|0.0     |4.297     |1962-01-01  |0.0    |Sherlock Holmes and the Deadly Necklace|
|43011|0.0     |4.417     |1962-01-01  |0.0    |Sodom and Gomorrah                     |
+-----+--------+----------+------------+-------+---------------------------------------+
only showing top 10 rows
```

Figure 2-12. *Glimpse of data after changing the datatype*

In the preceding output, if you observe closely, there is a visible difference in how the values of the `budget` and `revenue` columns are displayed with a `.0` value, indicating the datatype. Compare Figure 2-4 with Figure 2-12 to observe these differences before and after casting.

Descriptive Statistics

To analyze any data, you should have a keen understanding of the type of data, its distribution, and its dispersion. Spark has a nice suite of built-in functions that will make it easier to quickly calculate these fields. The `describe` function in Spark is very handy, as it gives the count of total non-missing values for each column, mean/average, standard deviation, and minimum and maximum values (see Figure 2-13). You can transform the output of this DataFrame and use it later in missing-value imputations. We discuss this in more depth in the model-building section.

```
df.describe()
```

```
+-------+------------------+--------------------+-----------------+--------------------+--------------------+
|summary|                id|              budget|       popularity|             revenue|               title|
+-------+------------------+--------------------+-----------------+--------------------+--------------------+
|  count|             43784|               43873|            43783|               43783|               43694|
|   mean|44502.304312077475|   3736901.834963166|5.295444259579189|   9697079.597382545|            Infinity|
| stddev|27189.646588626343|1.5871814952777334E7|6.168030519208248|5.6879384496288106E7|                 NaN|
|    min|                 2|                 0.0|              0.6|                 0.0|!Women Art Revolu...|
|    max|            100988|               3.8E8|            180.0|        2.78796518E9|       시크릿 Secret|
+-------+------------------+--------------------+-----------------+--------------------+--------------------+
```

Figure 2-13. *Describe function output*

Mean presented here is arithmetic mean, which is the central value of a discrete set of observations. Mean is a good estimate for many operations but can be susceptible to the influence of outliers. Outliers have the potential to significantly impact the mean value. In contrast, the median provides a better estimate, as it is not influenced by outliers. However, calculating the median can be computationally expensive, especially due to the need for sorting.

In Spark, this operation can be resource-intensive due to sorting. To make this calculation faster, Spark implemented a variant of the Greenwald-Khanna algorithm with some speed optimizations. The function used for calculating the median is `approxQuantile`.

Three parameters have to be passed through the `approxQuantile` function, as follows:

- `col` – The name of the numerical column.
- `probabilities` – A list of quantile probabilities. Each number must belong to [0, 1]. For example, 0 is the minimum, 0.5 is the median, 1 is the maximum.

- `relativeError` – The relative target precision to achieve (>= 0). If set to 0, the exact quantiles are computed, which could be very expensive. Note that values greater than 1 are accepted but give the same result as 1.

Let's try calculating the median for the `budget` column.

```
#Since unknown values in budget are marked to be 0, let's filter out those
values before calculating the median
df_temp = df.filter((df['budget']!=0)&(df['budget'].isNotNull()) &
(~isnan(df['budget'])))
#Here the second parameter indicates the median value, which is 0.5; you
can also try adjusting the value to calculate other percentiles
median=df.approxQuantile('budget',[0.5],0.1)
#Printing the Value
print ('The median of budget is '+str(median))
```

Output:

```
The median of budget is [7000000.0]
```

If you wanted to calculate multiple columns' medians at once, you would just need to change the first parameter to `list`.

Unique/Distinct Values and Counts

You may sometimes just want to know the number of levels (cardinality) within a variable. You can determine this using the `countDistinct` function available in Spark (see Figure 2-14).

```
# Counts the distinct occurances of titles
df.agg(countDistinct(col("title")).alias("count")).show()
```

```
+-----+
|count|
+-----+
|41138|
+-----+
```

Figure 2-14. *Distinct titles count output*

Alternatively, you can also look into the distinct occurrences using the `distinct` function (see Figure 2-15).

```
# Counts the distinct occurrences of titles
df.select('title').distinct().show(10,False)
```

```
+-------------------------------------------+
|title                                      |
+-------------------------------------------+
|The Corn Is Green                          |
|Meet The Browns - The Play                 |
|Morenita, El Escandalo                     |
|Father Takes a Wife                        |
|The Werewolf of Washington                 |
|My Wife Is a Gangster                      |
|Depeche Mode: Touring the Angel Live in Milan|
|A Woman Is a Woman                         |
|History Is Made at Night                   |
|Colombian Love                             |
+-------------------------------------------+
only showing top 10 rows
```

Figure 2-15. *Distinct titles output*

Intuitively, you may also want to check out distinct titles by year. So, let's first extract the year from the release date.

```
# Extracting year from the release date
df_temp=df.withColumn('release_year',year('release_date'))
```

You should be able to extract the month and day from the `date` variable using similar functions if needed. The following two commands are shown for demonstration purposes and are not required for the desired output. The third command gets you the distinct counts by year (see Figure 2-16).

```
# Extracting month
df_temp=df_temp.withColumn('release_month',month('release_date'))
# Extracting day of month
df_temp=df_temp.withColumn('release_day',dayofmonth('release_date'))
# Calculating the distinct counts by the year
df_temp.groupBy("release_year").agg(countDistinct("title")).show(10,False)
```

```
+------------+------------+
|release_year|count(title)|
+------------+------------+
|1959        |271         |
|1990        |496         |
|1975        |365         |
|1977        |415         |
|1924        |19          |
|2003        |1199        |
|2007        |1896        |
|2018        |4           |
|1974        |434         |
|2015        |13          |
+------------+------------+
only showing top 10 rows
```

Figure 2-16. *Distinct titles count by year*

Again, you should be able to sort the output by descending or ascending order using the functions discussed in the sorting section.

Filtering

Spark offers multiple ways to filter data. In fact, you encountered the `filter` function even before getting into this section. It demonstrates how important this function is even in basic operations. You may also encounter the `where` function when filtering. Both of these functions work the same way. However, `filter` is the standard Scala name for such a function, and `where` is for people who prefer SQL.

```
.where() = .filter()
```

Earlier, you encountered conditions in `filter` that included `or` and `and` conditions. In this demonstration, let's try other available methods to filter the data. Say you would like to filter all the titles that start with `Meet`; how should you approach this? If you are a SQL user, you can do this simply by using a `like` condition, as shown here. PySpark offers regex functions that can be used for this purpose (see Figure 2-17).

```
df.filter(df['title'].like('Meet%')).show(10,False)
```

```
+-----+---------+----------+------------+-----------+--------------------------+
|id   |budget   |popularity|release_date|revenue    |title                     |
+-----+---------+----------+------------+-----------+--------------------------+
|43957|500000.0 |2.649     |2005-06-28  |1000000.0  |Meet The Browns - The Play|
|39997|0.0      |3.585     |1989-11-15  |0.0        |Meet the Hollowheads      |
|16710|0.0      |11.495    |2008-03-21  |4.1939392E7|Meet the Browns           |
|20430|0.0      |3.614     |2004-01-29  |0.0        |Meet Market               |
|76435|0.0      |1.775     |2011-03-31  |0.0        |Meet the In-Laws          |
|76516|5000000.0|4.05      |1990-11-08  |485772.0   |Meet the Applegates       |
|7278 |3.0E7    |11.116    |2008-01-24  |8.4646832E7|Meet the Spartans         |
|32574|0.0      |7.42      |1941-03-14  |0.0        |Meet John Doe             |
|40506|0.0      |4.814     |1997-01-31  |0.0        |Meet Wally Sparks         |
|40688|2.4E7    |6.848     |1998-03-27  |4562146.0  |Meet the Deedles          |
+-----+---------+----------+------------+-----------+--------------------------+
only showing top 10 rows
```

Figure 2-17. *Filtering matches using a "like" expression*

Now, let's find the titles that *do not* end with an "s" (see Figure 2-18).

```
df.filter(~df['title'].like('%s')).show(10,False)
```

```
+-----+--------+----------+------------+-------+-------------------------------------+
|id   |budget  |popularity|release_date|revenue|title                                |
+-----+--------+----------+------------+-------+-------------------------------------+
|43000|0.0     |2.503     |1962-05-23  |0.0    |The Elusive Corporal                 |
|43001|0.0     |5.51      |1962-11-12  |0.0    |Sundays and Cybele                   |
|43002|0.0     |5.62      |1962-05-24  |0.0    |Lonely Are the Brave                 |
|43003|0.0     |7.159     |1975-03-12  |0.0    |F for Fake                           |
|43004|500000.0|3.988     |1962-10-09  |0.0    |Long Day's Journey Into Night        |
|43006|0.0     |3.194     |1962-03-09  |0.0    |My Geisha                            |
|43007|0.0     |2.689     |1962-10-31  |0.0    |Period of Adjustment                 |
|43008|0.0     |6.537     |1959-03-13  |0.0    |The Hanging Tree                     |
|43010|0.0     |4.297     |1962-01-01  |0.0    |Sherlock Holmes and the Deadly Necklace|
|43011|0.0     |4.417     |1962-01-01  |0.0    |Sodom and Gomorrah                   |
+-----+--------+----------+------------+-------+-------------------------------------+
only showing top 10 rows
```

Figure 2-18. *Filtering non-matches using a "like" expression*

If you wanted to find any title that contains "ove," you could use the `rlike` function, which is a regular expression (regex; see Figure 2-19).

```
df.filter(df['title'].rlike('\w*ove')).show(10,False)
```

```
+-----+------+----------+------------+------------+-----------------------+
|id   |budget|popularity|release_date|revenue     |title                  |
+-----+------+----------+------------+------------+-----------------------+
|43100|0.0   |7.252     |1959-10-07  |0.0         |General Della Rovere   |
|43152|0.0   |5.126     |2001-06-21  |0.0         |Love on a Diet         |
|43191|0.0   |4.921     |1952-08-29  |0.0         |Beware, My Lovely      |
|43281|0.0   |2.411     |1989-11-22  |0.0         |Love Without Pity      |
|43343|0.0   |3.174     |1953-12-25  |0.0         |Easy to Love           |
|43347|3.0E7 |14.863    |2010-11-22  |1.02820008E8|Love & Other Drugs     |
|43362|0.0   |1.705     |1952-02-23  |0.0         |Love Is Better Than Ever|
|43363|0.0   |2.02      |1952-05-29  |0.0         |Lovely to Look At      |
|43395|0.0   |4.758     |1950-11-10  |0.0         |Two Weeks with Love    |
|43455|0.0   |4.669     |1948-08-23  |0.0         |The Loves of Carmen    |
+-----+------+----------+------------+------------+-----------------------+
only showing top 10 rows
```

Figure 2-19. *Filtering matches using a regular expression*

The preceding expression can also be rewritten as follows:

```
df.filter(df.title.contains('ove')).show()
```

There will be situations where you'll have thousands of columns and want to identify or subset the columns by a particular prefix or suffix. You can achieve this using the `colRegex` function. First, identify the variables that start with "`re`" (see Figure 2-20).

```
df.select(df.colRegex("`re\w*`")).printSchema()
```

```
root
 |-- release_date: date (nullable = true)
 |-- revenue: float (nullable = true)
```

Figure 2-20. *Filtering columns by prefix using regular expressions*

Okay, so how can you identify variables that end with a particular suffix? Here you have three variables that end with "e"; let's see if we can identify them by tweaking the earlier regular expressions (see Figure 2-21).

```
df.select(df.colRegex("`\w*e`")).printSchema()
```

```
root
 |-- release_date: date (nullable = true)
 |-- revenue: float (nullable = true)
 |-- title: string (nullable = true)
```

Figure 2-21. *Filtering columns by suffix using regular expressions*

Regex can be confusing, so let's take a brief refresher on the topic. In general, you can identify all the characters and numerics by metacharacters. Regex uses these metacharacters to match any literal or alphanumeric expression. Here is a short list of the most used metacharacters.

Single characters:

- `\d` – Identifies numbers between 0 and 9
- `\w` – Identifies all upper- and lowercase alphabets and numbers from 0 to 9 [A–Z a–z 0–9]
- `\s` – Whitespace
- `.` – Any character

You also have a set of quantifiers that guide how many characters to search, as follows.

Quantifiers:

- * – 0 or more characters
- + – 1 or more characters
- ? – 0 or 1 characters
- `{min,max}` – Specifies the range, any characters between the range including `min, max`
- `{n}` – Exactly *n* characters

If you look back at both of the earlier expressions, you will see that we have used `\w,` indicating that we want to match all alphanumeric characters, ignoring the case. In the expression where we wanted to identify variables with the "re" prefix, `\w` was followed by the quantifier * to indicate that we wanted to include all characters after "re."

Creating New Columns

Creating new features (columns) plays a critical role in many analytical applications. Spark offers multiple ways in which you can create new columns. The simplest method is through the `withColumn` function. Let's say you want to calculate the variance of popularity. Variance is a measure of how far the numbers are spread from the mean and is represented by following formula:

$$S^2 = \sum\left(X_i - \bar{X}\right)^2 / (n-1)$$

x_i – Individual observations

x – Mean

n – Total number of observations

So, let's first calculate the mean by using the following command. The agg function used here is better than using describe when you are looking for a specific statistic:

```
# Calculate the mean popularity
mean_pop=df.agg({'popularity': 'mean'}).collect()[0]['avg(popularity)']
count_obs= df.count()
```

Let's add this to all the rows since we need this value along with individual observation values using the withColumn and lit functions. The lit function is a way to interact with column literals. It is very useful when you want to create a column with a value directly.

```
df=df.withColumn('mean_popularity',lit(mean_pop))
```

The mean_popularity column has the same value for all the rows. The pow function in the following command helps you raise the number to a power. Here, you need to square the number, hence 2 is passed as the parameter.

```
#Calculate the variance of the 'popularity' column
df=df.withColumn('varaiance',pow((df['popularity']-df['mean_
popularity']),2))
variance_sum=df.agg({'varaiance': 'sum'}).collect()[0]['sum(varaiance)']
```

The preceding command sums up all the differences across all the rows. And in the final step, we divide the sum by the number of observations, which yields the result.

```
variance_population= variance_sum/(count_obs-1)
```

Output:

```
37.85868805766277
```

Although this variance calculation involved multiple steps, the idea is to give you a flavor of operations that might be involved in creating a new feature. Once you have the value of a variance, you can create custom variables based on it.

Let's step back. You may ask, "If I have multiple variables to create, how can I do it all at once and keep them well structured?" We walk you through the solution next. Say you would like to create variables that can classify both the `budget` and the `popularity` variables into high, medium, and low. First, you define what the high, medium, and low thresholds are based on. The following thresholds are used for convenience; you can alter them to values based on more rigorous analysis.

The following piece of code defines a simple Python function. It is passing two variables into the function, and based on the threshold, it decides to return the new variables, which are both strings.

```
def new_cols(budget,popularity):
 if budget<10000000: budget_cat='Small'
 elif budget<100000000: budget_cat='Medium'
 else: budget_cat='Big'
 if popularity<3: ratings='Low'
 elif popularity<5: ratings='Mid'
 else: ratings='High'
 return budget_cat,ratings
```

Now, here is the fun part: we are now using user-defined functions. So, what are user-defined functions? If you are coming from the Python world, these are very similar to the `.map()` and `.apply()methods` for the Pandas DataFrame and series. In simple words, user-defined functions allow developers to use values from rows in a DataFrame as input and can map it to the entire DataFrame. One crucial fact is that you must specify the output datatype. Unfortunately, there is no getting around this. This can be painful when you have to write out multiple columns. We discuss another method after this that can be used as an alternative.

In the command, you define the user-defined function. Observe that we create `StructType` to hold the new variables. We also explicitly define what the datatype is that will be returned. We know both of the return types are strings, so we use `StringType()`. `StructType` objects define the schema of Spark DataFrames. `StructType` objects contain a list of `StructField` objects that define the name, type, and nullable flag for each column in a DataFrame.

`StructType` objects are helpful for eliminating any order dependencies in your code. What do we mean by that? If you have multiple functions that need to run before the final step, you can wrap them all under the `StructType` to return an array of

results rather than create multiple columns. This is useful for reading and managing long codebases that involve various transformations and actions. For this example, we will split the StructType that holds multiple outputs into individual columns for further access.

```
# Apply the user-defined function on the DataFrame
udfB=udf(new_cols,StructType([StructField("budget_cat", StringType(),
True),StructField("ratings", StringType(), True)]))
```

The following command passes a user-defined function with two input columns: budget and popularity:

```
temp_df=df.select('id','budget','popularity').withColumn("newcat",udfB("bud
get","popularity"))
# Unbundle the struct type columns into individual columns and drop the
struct type
df_with_newcols = temp_df.select('id', 'budget',' popularity', 'newcat').
withColumn('budget_cat', temp_df.newcat.getItem('budget_cat')).
withColumn('ratings', temp_df.newcat.getItem('ratings')).drop('newcat')
df_with_newcols.show(15,False)
```

In the following command, note that we are again using the withColumn function to extract data inside the StructType. Since we have the StructType holding string datatypes, we use the getItem function to access the elements within the StructType. Observe that we are concatenating the functions one after the other to achieve the desired output in a single line (see Figure 2-22).

```
+-----+---------+----------+----------+-------+
|id   |budget   |popularity|budget_cat|ratings|
+-----+---------+----------+----------+-------+
|43000|0.0      |2.503     |Small     |Low    |
|43001|0.0      |5.51      |Small     |High   |
|43002|0.0      |5.62      |Small     |High   |
|43003|0.0      |7.159     |Small     |High   |
|43004|500000.0 |3.988     |Small     |Mid    |
|43006|0.0      |3.194     |Small     |Mid    |
|43007|0.0      |2.689     |Small     |Low    |
|43008|0.0      |6.537     |Small     |High   |
|43010|0.0      |4.297     |Small     |Mid    |
|43011|0.0      |4.417     |Small     |Mid    |
|43012|7000000.0|4.722     |Small     |Mid    |
|43013|0.0      |2.543     |Small     |Low    |
|43014|0.0      |4.303     |Small     |Mid    |
|43015|0.0      |3.493     |Small     |Mid    |
|43016|0.0      |2.851     |Small     |Low    |
+-----+---------+----------+----------+-------+
only showing top 15 rows
```

Figure 2-22. *Output of newly created DataFrame with custom columns*

Another way you can achieve the same result is using the `when` function. One advantage of using this function is you don't have to define the output datatype. This is handy for quick and dirty operations. Let's re-create the preceding columns using the `when` function.

```
df_with_newcols = df.select('id', 'budget',' popularity').\
withColumn('budget_cat', when(df['budget']<10000000,'Small').when(df
['budget']<100000000,'Medium').otherwise('Big')).\
withColumn('ratings', when(df['popularity']<3,'Low').
when(df['popularity']<5,'Mid').otherwise('High'))
```

If you observe the preceding command, you can see that we concatenated many conditions to define our new variables. This produces the same output as the previous way. Another helpful tip: Using a combination of `withColumn` and `when` functions, you can update the value of an existing column if you use the same column name. You don't have to create a new temporary DataFrame (as shown) for the operations. Instead, you can utilize the same one to update or add column values.

Deleting and Renaming Columns

You can always drop a column or columns using the drop function.

```
columns_to_drop=['budget_cat']
df_with_newcols=df_with_newcols.drop(*columns_to_drop)
```

You can verify the output by using the printSchema function.

Renaming can be done using either the withColumnRenamed function or the alias function.

```
df_with_newcols = df_with_newcols.withColumnRenamed('id','film_id') .with
ColumnRenamed('ratings','film_ratings')
```

If you want to change multiple column names, you can try the following command:

```
# You can define all the variable changes in the list
new_names = [('budget','film_budget'),('popularity','film_popularity')]
# Applying the alias function
df_with_newcols_renamed = df_with_newcols.select(list(map(lambda
old,new:col(old).alias(new),*zip(*new_names))))
```

Both of the preceding methods have the same execution plan. You can verify all these outputs by using the printSchema function.

Use the following dataset for Exercise 2-1: https://www.kaggle.com/kakarlaramcharan/tmdb-data-0920.

Pandas API on Spark

Before we close out the chapter, we explore the new features added to Spark in recent releases. One of the revelations is the integration of the Pandas API on Spark. This can be a game changer for those coming in from the pandas world. With one line of code, you can use the distributed computing power of Spark using familiar methods. It is also noteworthy that pandas on Spark is much faster than DASK. If you are unaware of DASK, it is an open-source Python library for parallel computing.

Note that when working with pandas on Spark, the pyspark.pandas.frame.DataFrame class is used. The pandas-on-Spark DataFrame and the Spark DataFrame are virtually interchangeable. Let's see the variants of DataFrames (see Figure 2-23). If your dataframe class starts with pyspark, you can be confident that it is using the parallel

computing that is configured. If not, you may change it to Spark dataframe or pandas on Spark dataframe. Do not confuse this with a core pandas dataframe that runs only on a single machine.

```
# Dataframe DataTypes:
df_with_newcols_psd_multi_machine = df_with_newcols.pandas_api()
df_with_newcols_pandas_core_single_machine=df_with_newcols.toPandas()
print("Distributed: Multi Machine")
print(f'df_with_newcols - spark dataframe : {type(df_with_newcols)}')
print(f'df_with_newcols_psd_multi_machine - pandas on spark: {type(df_with_
newcols_psd_multi_machine)}')
print("Non Distributed: Single Machine")
print(f'df_with_newcols_pandas_core_single_machine - core Pandas: {type(df_
with_newcols_pandas_core_single_machine)}')
```

```
Distributed: Multi Machine
df_with_newcols - spark dataframe : <class 'pyspark.sql.dataframe.DataFrame'>
df_with_newcols_psd_multi_machine - pandas on spark: <class 'pyspark.pandas.frame.DataFrame'>
Non Distributed: Single Machine
df_with_newcols_pandas_core_single_machine - core Pandas: <class 'pandas.core.frame.DataFrame'>
```

Figure 2-23. *Types of dataframes*

To illustrate how you can use the `pandas on spark` DataFrame class, an example is provided here (see Figures 2-24 and 2-25):

```
#Pandas on spark dataframe
import pyspark.pandas as ps
df_pd_distributed=ps.read_csv("movie_data_part1.csv",sep="|")
df_pd_distributed[['budget','original_title','popularity']].head()
```

	budget	original_title	popularity
0	0.0	Le Caporal épinglé	2.503
1	0.0	Cybèle ou les dimanches de ville d'Avray	5.510
2	0.0	Lonely Are the Brave	5.620
3	0.0	Vérités et Mensonges	7.159
4	500000.0	Long Day's Journey Into Night	3.988

Figure 2-24. *Output of reading a csv via the Pandas API*

```
#Verify dataframe datatype
print(f'DataFrame DataType: {type(df_pd_distributed)}')
```

```
DataFrame DataType: <class 'pyspark.pandas.frame.DataFrame'>
```

Figure 2-25. *Output of datatype of dataframe*

In most scenarios, typecasting between PySpark and the Pandas API on Spark is automatic when you call the API. You can compare both of the DataFrames before and after conversion using the `dtypes` method. The same is true when converting Pandas API on Spark to PySpark.

The typecasting between Pandas API on Spark and pandas is also very similar. Note that certain datatypes are exclusive to only to pandas, which are not planned to be incorporated into Spark yet.

- `pd.SparseDtype`
- `pd.DatetimeTZDtype`
- `pd.UInt*Dtype`
- `pd.BooleanDtype`
- `pd.StringDtype`

The `transform` and `apply` methods are available for Pandas API on Spark for column manipulations. Transform expects to return the same length of rows as input dataframe, whereas apply methods can return an arbitrary number of rows. In addition to these methods, `pandas_on_spark.transform_batch` and `pandas_on_spark.apply_batch` are also available. These batch methods allow you to call a function that takes and returns a pandas DataFrame. When you use these batch methods, Spark iterates on a large DataFrame, breaks it into batches, and calls the function you specify per batch. Be cautious when applying these methods on larger datasets; **it is recommended that the indexes of all batch DataFrames are aligned**. You can spend a significant amount of time debugging this error. To avoid this, align the indexes of columns as a precursor.

```
# Applying transformations on columns
import pyspark.pandas as ps

def pandas_plus(pser):
    return (pser+1)  # allows an arbitrary length
```

```
df_pd_distributed_func=df_pd_distributed[['budget','popularity']].
transform(pandas_plus)
df_pd_distributed_func.head()
type(df_pd_distributed_func)
```

```
/usr/local/spark/python/pyspark/pandas/utils.py:1016: PandasAPIOnSparkAdviceWarning: If the type hints is not specified for `transform`, it is expensive to infer the data type inter
nally.
  warnings.warn(message, PandasAPIOnSparkAdviceWarning)
```

	budget	popularity
0	1.0	3.503
1	1.0	6.510
2	1.0	6.620
3	1.0	8.159
4	500001.0	4.988

Figure 2-26. *Output of data transformation using transform*

In the previous example, the schema is inferred by taking the top rows of output. This is a simple example. Usual use cases may involve several expensive operations, such as shuffling in the upstream of the execution plan. Pandas API on Spark will end up executing the Spark job twice—once for schema inference and once for processing actual data with the schema.

To avoid the consequences, Pandas API on Spark has its own type-hinting style to specify the schema and avoid schema inference. Pandas API on Spark understands the type hints specified in the return type and converts it as a Spark schema for Pandas UDFs used internally. The way of type-hinting has evolved.

```
def pandas_plus(pser) ->ps.DataFrame[int,float]:
    return (pser+1)  # allows an arbitrary length

df_pd_distributed_func=df_pd_distributed[['budget','popularity']].
transform(pandas_plus)
df_pd_distributed_func.head()
type(df_pd_distributed_func)
```

	budget	popularity
0	1.0	3.503
1	1.0	6.510
2	1.0	6.620
3	1.0	8.159
4	500001.0	4.988

Figure 2-27. *Output of data transformation using transform*

Observe that the warning disappeared as soon as the type-hinting was included in the function definition. It is best practice to have type hints to improve the performance of your operations, especially with Pandas API on Spark.

Pandas API on Spark is an integration layer, and users must be conscious to avoid unintended inefficiencies. Some of the best practices to incorporate are

- Use execution plans to understand how the data will be processed; you can see the execution plan by calling the `.spark.explain()` method.
- After many operations on Pandas API on Spark objects, the underlying Spark planner can slow down due to the vast and complex plan. If the Spark plan becomes huge or if it takes a long time, `DataFrame.spark.checkpoint()` or `DataFrame.spark.local_checkpoint()` can be helpful.
- Due to data exchange, operations like sorting values are more challenging to execute in parallel or distributed environments than in a single memory machine.
- Avoid computations on a single partition. If you are using rank without specifying a partition, you may put all the data into a single partition on a single machine.
- Avoid reserved or duplicate column names.
- Pandas API on Spark disallows the operations on different DataFrames (or Series) by default to prevent expensive operations. Reduce the operations on different DataFrames or Series.

EXERCISE 2-1: DATA MANIPULATIONS

Question 1: Identify the titles that were repeated between 2000 and 2015 and count the number of titles. (Hint: Use a filter.)

Question 2: Identify all titles that contain Harry in the title name.

Question 3: Create a new column as a binary indicator of whether the original language is English.

Question 4: Tabulate the mean of popularity by year.

Summary

- You got an overview of the Spark ecosystem and learned how PySpark fits in this environment, and you also learned the advantages of using Spark.
- You now know the differences between RDDs, DataFrames, and datasets.
- You ventured into data manipulations and learned to use various built-in functions.
- You learned about the user-defined function and saw its implementation and mechanics.
- You learned about Pandas API on Spark, its implementation, and best practices.

Good job! You are now familiar with some key concepts useful throughout your journey in this book. In the next chapter, we take a deep dive into some of the intermediate and advanced data manipulation techniques.

CHAPTER 3

Utility Functions and Visualizations

In this chapter, we explore the utility of several advanced functions available in PySpark. To enhance your understanding, we recommend reviewing the previous chapter and experimenting with the following operations on any dataset you choose. Our focus is on windowing functions and other topics essential for developing and applying Spark programs to large datasets. As we wrap up the chapter, we also introduce visualization and machine learning processes, preparing you to seamlessly transition into the subsequent machine learning chapters.

This chapter covers the following topics:

- Additional data manipulations
- Data visualizations
- Machine learning overview

Additional Data Manipulations

Let's start by reviewing some data manipulations not covered in the previous chapter.

String Functions

For this section, we rerun the commands executed in Chapter 2.

```
# Rerunning the following command for recreating budget_cat variable
df_with_newcols = df.select('id', 'budget',' popularity').\
withColumn('budget_cat', when(df['budget']<10000000,'Small').when
(df['budget']<100000000,'Medium').otherwise('Big')).\
```


© Ramcharan Kakarla, Sundar Krishnan, Balaji Dhamodharan and Venkata Gunnu 2024
R. Kakarla et al., *Applied Data Science Using PySpark*, https://doi.org/10.1007/979-8-8688-0820-3_3

```
withColumn('ratings', when(df['popularity']<3,'Low').
when(df['popularity']<5,'Mid').otherwise('High'))
```

If you want to combine the values of `budget_cat` and `ratings` into a single column, you can do so using the `CONCAT` function. Additionally, you can change the case of the new column to lowercase and trim away any whitespace using the `lower` and `trim` functions, respectively (see Figure 3-1).

```
# Concatenating two variables
df_with_newcols=df_with_newcols.withColumn('BudgetRating_
Category',concat(df_with_newcols.budget_cat,df_with_newcols.ratings))
# Changing the new variable to lowercase
df_with_newcols=df_with_newcols.withColumn('BudgetRating_
Category',trim(lower(df_with_newcols.BudgetRating_Category)))
df_with_newcols.show()
```

```
+-----+---------+----------+----------+-------+--------------------+
|   id|   budget|popularity|budget_cat|ratings|BudgetRating_Category|
+-----+---------+----------+----------+-------+--------------------+
|43000|      0.0|     2.503|     Small|    Low|           smalllow|
|43001|      0.0|      5.51|     Small|   High|          smallhigh|
|43002|      0.0|      5.62|     Small|   High|          smallhigh|
|43003|      0.0|     7.159|     Small|   High|          smallhigh|
|43004| 500000.0|     3.988|     Small|    Mid|           smallmid|
|43006|      0.0|     3.194|     Small|    Mid|           smallmid|
|43007|      0.0|     2.689|     Small|    Low|           smalllow|
|43008|      0.0|     6.537|     Small|   High|          smallhigh|
|43010|      0.0|     4.297|     Small|    Mid|           smallmid|
|43011|      0.0|     4.417|     Small|    Mid|           smallmid|
|43012|7000000.0|     4.722|     Small|    Mid|           smallmid|
|43013|      0.0|     2.543|     Small|    Low|           smalllow|
|43014|      0.0|     4.303|     Small|    Mid|           smallmid|
|43015|      0.0|     3.493|     Small|    Mid|           smallmid|
|43016|      0.0|     2.851|     Small|    Low|           smalllow|
|43017|      0.0|     4.047|     Small|    Mid|           smallmid|
|43018|      0.0|     2.661|     Small|    Low|           smalllow|
|43019|      0.0|     3.225|     Small|    Mid|           smallmid|
|43020|      0.0|      5.72|     Small|   High|          smallhigh|
|43021|      0.0|     3.292|     Small|    Mid|           smallmid|
+-----+---------+----------+----------+-------+--------------------+
only showing top 20 rows
```

Figure 3-1. *Output of string functions*

Registering DataFrames

So far, you've been working with DataFrames and their functions. Wouldn't it be fantastic to leverage SQL functions with these DataFrames? Luckily, PySpark allows you to run SQL queries on DataFrames. Spark handles all the operations. To do this, you need to register the DataFrame as a temporary table using the `createOrReplaceTempView` function, which creates an in-memory table. Keep in mind that this table is only accessible within the Spark session in which it was created, and the data is stored in Hive's columnar format. This setup lets you access all columns just like a regular SQL table. You can also overwrite the existing temporary table with different data. Try registering the DataFrame as a temporary table and see how simple SQL queries can achieve the desired results without using any functions (as shown in Figure 3-2a).

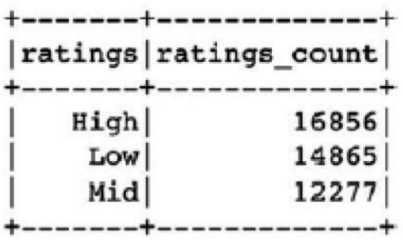
```
# Registering temporary table
df_with_newcols.createOrReplaceTempView ('temp_data')
# Applying the function to show the results
spark.sql('select ratings, count(ratings) from temp_data group by
ratings').show(False,10)
```

```
+-------+-------------+
|ratings|ratings_count|
+-------+-------------+
|   High|        16856|
|    Low|        14865|
|    Mid|        12277|
+-------+-------------+
```

Figure 3-2a. *Output of Spark SQL output on a temp table*

In the latest version of Spark, you can skip this step and directly query the DataFrame in memory using `spark.sql` without registering. You can find the result is precisely the same (see Figure 3-2b).

```
# Using spark latest capability
spark.sql('select ratings, count(ratings) as ratings_count from {temp_df}
group by ratings',temp_df=df_with_newcols).show()
```



ratings	ratings_count
High	16856
Low	14865
Mid	12277

Figure 3-2b. *Output of Spark SQL on an existing dataframe*

Alternate query with dynamic insertion using string formatting:

```
spark.sql('SELECT ratings, COUNT(ratings) AS ratings_count FROM {} GROUP BY
ratings'.format(temp_df)).show()
```

Window Functions

Window functions allow you to perform calculations across a set of rows relative to a current row. They are particularly useful for performing time-based analytical functions. Much of feature engineering relies on window functions. These functions can be categorized into three types:

1. **Aggregate Functions:** These group multiple rows to produce a single summary value, such as sum, average, min, max, or count.
2. **Ranking Functions**
3. **Analytical Functions**

When using a window function, you need to specify three key components:

- **Partition**: Determines which rows are included in the same partition as the current row. For example, to calculate the top budget movies by year, you might partition by year.
- **Order**: Defines how to order the rows within each partition, either ascending or descending.
- **Frame**: Specifies how many rows to include above or below the current row, based on its relative position.

PySpark includes a built-in set of functions for ranking and analytical operations. Table 3-1 provides a concise and useful list of these functions.

Table 3-1. *Types of Windowing Functions*

Type	Functions
Ranking	rank
	denseRank
	percentRank
	rowNumber
	ntile
Analytical	lead
	lag
	lastValue
	firstValue
	cumeDist

You will gain a better understanding of these concepts with the following illustrations. Let's calculate the deciles of popularity. What do we mean by deciling? If you come from an analytical background, you likely already appreciate deciles. Deciling is the process of dividing the distribution into ten equal buckets, either in ascending or descending order. These are initial steps in evaluating the performance of a predictive model. We delve deeper into model performance and validation statistics in later chapters. For now, let's see how to calculate the deciles for popularity.

```
# Importing the window functions
from pyspark.sql.window import *
# Step 1: Filtering the missing values
df_with_newcols=df_with_newcols.filter( (df_with_newcols['popularity'].
isNotNull()) & (~isnan(df_with_newcols['popularity'])) )
# Step 2: Applying the window functions for calculating deciles
df_with_newcols = df_with_newcols.select("id","budget","popularity",
ntile(10).over(Window.partitionBy().orderBy(df_with_newcols['popularity'].
desc()*))).alias("decile_rank"))
```

```
# Step 3:Dispalying the values
df_with_newcols.groupby("decile_rank").agg(min('popularity').alias
('min_popularity'),max('popularity').alias('max_popularity'),count
('popularity')).show()
```

Let's break down what's happening in Step 2. We use the `ntile` function, which divides the variable distribution into *n* buckets. Additionally, we specify how we would like to split the distribution using a partition. In this case, since we did not mention any partition for the data, it will inherently use all the data. We then utilize an `orderBy` to arrange all the rows in the data by popularity value in descending order. Finally, the `alias` function is used to rename the decile column. Since Spark operates on lazy evaluation, executing this command will not immediately return the result.

Moving to Step 3, we are performing an action. As we execute this step, underlying plans are executed and deciles are calculated. In Step 3, we gather all the statistics we want to use in the agg function from the grouped DataFrame. Additionally, we include the aggregate function's minimum, maximum, and count values. We expect to see approximately equal numbers of records in each decile, and we add the `count` to verify this fact. The result is shown in Figure 3-3.

decile_rank	min_popularity	max_popularity	count(popularity)
1	10.185	180.0	4379
2	7.481	10.182	4379
3	5.841	7.481	4379
4	4.823	5.841	4378
5	4.054	4.822	4378
6	3.383	4.054	4378
7	2.747	3.383	4378
8	2.075	2.747	4378
9	1.389	2.075	4378
10	0.6	1.389	4378

Figure 3-3. *Output of popularity deciles*

We now have a fair idea of how popularity ratings vary across ten equally distributed buckets. Let's say you want to know what the second most popular movie was in 1970. How can you find it using the functions you just learned?

Before jumping in, use the DataFrame in memory. What if I have already overwritten the data? Okay, at this point, if you have overwritten your DataFrame, you can always reload the data. However, remember to cast the variables to the appropriate datatypes.

```
# Step 1: Import the window functions
from pyspark.sql.window import *
# Step 2: Select the required subset of columns
df_second_best = df.select('id', 'popularity', 'release_date')
# Step 3: Create the year column from release date
df_second_best=df_second_best.withColumn('release_year',year
('release_date')).drop('release_date')
# Step 4: Define partition function
year_window = Window.partitionBy(df_second_best['release_year']).
orderBy(df_second_best['popularity'].desc())
# Step 5: Apply partition function
df_second_best=df_second_best.select('id', 'popularity',
'release_year',rank().over(year_window).alias("rank"))
# Step 6: Find the second-best rating for the year 1970
df_second_best.filter((df_second_best['release_year']==1970) &
(df_second_best['rank']==2)).show()
```

Let's rewind and observe what's happening in the preceding steps. First, we selected the required columns and extracted the year from the release date. You must be familiar with this step from the earlier section. In Step 4, we defined our window function. We want to partition the data by year. The reason for doing this is we want to know the second-highest rating in the year 1970. This partition will allow us to rank all popularity ratings in the data by year. In Step 5, we used a `rank` function along with the `window.partitionBy` function we defined earlier to arrive at the desired result. Wait—we are not done yet! We want to identify the results for the year 1970, so we filter the data to the year 1970 and subset the data where the row rank is 2, as illustrated in Step 6. The results are shown in Figure 3-4.

```
+-----+----------+------------+----+
|   id|popularity|release_year|rank|
+-----+----------+------------+----+
|11202|    14.029|        1970|   2|
+-----+----------+------------+----+
```

Figure 3-4. *Output of the second highest popularity rating in the year 1970*

We used the `rank function` here. There is a subtle difference between `rank` and `denseRank` — `rank` gives the ranking within your ordered partitions, and ties are assigned the same rank. The ranks consecutive to ties are given a higher rank. If you have two rows with rank 2, the next rank will be 4. In `denseRank`, no ranks are skipped and they are consecutive. You have to choose these functions based on your requirements.

Let's take this a step further. What if your question was, what is the difference between the revenue of the year's highest-grossing film and other films within that year? You need to calculate the difference in revenue for a film after identifying the highest-grossing movie of that year. The following code demonstrates how to achieve this. The results are shown in Figure 3-5.

```
# Step 1: Import the window functions
from pyspark.sql.window import *
# Step 2: Select the required subset of columns
df_revenue = df.select('id', 'revenue', 'release_date')
# Step 3: Create the year column from the release date
df_revenue=df_revenue.withColumn('release_year',year('release_date')).
drop('release_date')
# Step 4: Define partition function along with range
windowRev =Window.partitionBy(df_revenue['release_year']).orderBy
(df_revenue['revenue'].desc()).rangeBetween(-sys.maxsize, sys.maxsize)
# Step 5: Apply partition function for the revenue differences
revenue_difference = (max(df_revenue['revenue']).over(windowRev) -
df_revenue['revenue'])
# Step 6: Final data
df_revenue.select('id', 'revenue', 'release_year',revenue_difference.
alias("revenue_difference")).show(10,False)
```

```
+-----+---------+------------+------------------+
|id   |revenue  |release_year|revenue_difference|
+-----+---------+------------+------------------+
|665  |1.64E8   |1959        |0.0               |
|10882|5.1E7    |1959        |1.13E8            |
|239  |2.5E7    |1959        |1.39E8            |
|4952 |1.875E7  |1959        |1.4525E8          |
|15944|1.7658E7 |1959        |1.46342E8         |
|213  |1.3275E7 |1959        |1.50724992E8      |
|27029|1.28E7   |1959        |1.512E8           |
|29996|1.22E7   |1959        |1.518E8           |
|11571|1.0E7    |1959        |1.54E8            |
|9660 |6800000.0|1959        |1.572E8           |
+-----+---------+------------+------------------+
only showing top 10 rows
```

Figure 3-5. *Output of revenue differences with the highest revenue by year*

You'll notice not many changes from the previous example until Step 3. In Step 4, alongside the `window.partitionBy` functions, we've incorporated the `rangeBetween` function. This function is useful for defining a boundary between the start and end (both inclusive). Both parameters within the `rangeBetween` function indicate that the lowest and highest possible values within that partition should be included. Transitioning to Step 5, we identify the highest values over the window function and subtract them from the row values. This yields the expected output, as demonstrated in Step 6.

Window functions can be an expensive topic. However, the preceding examples should provide you with a good idea of the possibilities of how you can utilize these functions. Most SQL windowing functions are available as built-in functions in PySpark.

Other Useful Functions

In this section, we try to uncover other valuable functions that we didn't mention in the earlier sections. They can be very effective based on the use cases.

Collect List

We know from previous analysis that some of the titles are repeated. Let's say you want to find all the years where the title "The Lost World" is repeated. We have a handy function to do so called `collect_list`. For this demonstration, we will use the existing DataFrame. The previous output is shown in Figure 3-6 for reference, while the new output is shown in Figure 3-7.

```
+--------------------+-----+
|title               |count|
+--------------------+-----+
|Les Misérables      |8    |
|The Three Musketeers|8    |
|Cinderella          |8    |
|Hamlet              |7    |
|The Island          |7    |
|Dracula             |7    |
|A Christmas Carol   |7    |
|Frankenstein        |7    |
|Framed              |6    |
|The Lost World      |6    |
+--------------------+-----+
only showing top 10 rows
```

Figure 3-6. *Output of filtered and sorted version with one-way frequency*

```
# Step 1: Create the year column from release date
df=df.withColumn('release_year',year('release_date'))
# Step 2: Apply collect_list function to gather all occurrences
df.filter("title=='The Lost World'").groupby('title').agg
(collect_list("release_year")).show(1,False)
```

```
+--------------+------------------------------------+
|title         |collect_list(release_year)          |
+--------------+------------------------------------+
|The Lost World|[1999, 2001, 1925, 1960, 1992, 1998]|
+--------------+------------------------------------+
```

Figure 3-7. *Output of collect_list*

If we break down Step 2, we filter out the title we are interested in and group the DataFrame by the title. This particular title has six occurrences. The `collect_list` function gathers the release years of this specific title, which is displayed in the output.

Sampling

Sampling holds a special place in analytics. Simply put, a *sample* is a subset of the population. But what exactly is a population? A *population* represents the entirety of data. It's crucial to be precise when considering the population you are interested in studying. For instance, if you define your population as all active users of Amazon, this

encompasses all users of Amazon regardless of region, language, or gender. However, in real-world scenarios, obtaining population data can be challenging and sometimes impractical.

We use samples to infer what is happening in the population. However, if sampling is not executed correctly, it can significantly impact your results. Broadly speaking, available sampling methods can be classified into probabilistic and non-probabilistic techniques. Probabilistic techniques involve random selection to choose the sample, whereas non-probabilistic techniques rely on subjective judgment. In data science, we typically employ probabilistic techniques to make informed decisions on samples. Some popular sampling techniques within probabilistic methods include simple random sampling, systematic random sampling, stratified random sampling, and cluster sampling.

In simple random sampling, every observation in the sample is randomly selected, ensuring that all observations have an equal chance of being chosen. There are two variants of simple random sampling.

- **Simple random sampling with replacement:** In this method, once an observation is randomly selected from the population, it is put back into the dataset before a second observation is selected. In other words, selection of one observation will not affect the outcome of other observations.
- **Simple random sampling without replacement:** In this method, once an observation is randomly selected from the population, it is not put back into the dataset before a second observation is selected. In other words, selection of one observation will dramatically affect the outcome of other observations.

In stratified sampling, you select the sample from a population that can be partitioned into subpopulations. For instance, suppose you know the proportion of the population with specific makes of cars in the luxury car segment. In that case, you can maintain the same proportion when selecting the sample using this technique.

Now, let's explore how to implement these techniques using PySpark.

```
# Simple random sampling in PySpark without replacement
df_sample = df.sample(False, 0.4, 11)
df_sample.count()
```

Output:

```
17711
```

The preceding command specifies three parameters. The first parameter, `True/False`, indicates whether you would like to conduct a sample with or without replacement. We've chosen `False` to indicate that we want to sample without replacement. The second parameter is the fraction, indicating the proportion of the population you would like to have in the sample. The third parameter is the seed, which ensures that you get the same result each time you run this snippet.

```
# Simple random sampling in PySpark with replacement
df_sample = df.sample(True, 0.4, 11)
df_sample.count()
```

Output:

```
17745
```

Note that the sample size is not exactly the ratio we requested in both cases. This is because Spark internally uses a Bernoulli sample technique for taking a sample. Because each element of the population is selected separately for the sample, the sample size is not fixed but rather follows a binomial distribution.

```
# Stratified sampling in PySpark
df_strat = df.sampleBy("release_year", fractions={1959: 0.2, 1960: 0.4,
1961: 0.4}, seed=11)
df_strat.count()
```

Output:

```
260
```

Stratified sample syntax is a bit different from the simple random sample. Notice that we are using the `sampleBy` function instead of `sample`. The first parameter here is the column you want to stratify. In this case, we are using the release year. For example, let's say we know that twice the number of movies were released in 1960 and 1961 compared to 1959; we would use that fraction to define our sample. This is represented in the second parameter.

Caching and Persisting

Caching is one of the most important concepts of Spark. When used correctly, it can save a lot of time when running large programs. But what is caching? It's a way to temporarily save data in memory, a practice common across different operating systems, software, and programming languages. So why is it useful? Well, if you can store your data in memory, you don't have to repeat computations every time the DataFrame is accessed, saving you the costs of expensive data I/O operations. We recommend using caching if you're going to access a DataFrame multiple times for multiple operations. If you're only going to use the DataFrame once or if the data can be computed easily, caching isn't necessary.

After reading your data, you can cache the first query on the DataFrame. The first query will run at the usual speed, but subsequent operations will benefit from enhanced computational speed. It's worth noting that caching is lazy, meaning your first operation on the DataFrame will take some time. You can cache any DataFrame using a cache function.

```
df.cache()
```

Persisting has a similar function or can be seen as a synonym for caching. They differ in the storage-level settings. Spark has five types of storage. This is obtained from the Spark documentation. See Table 3-2.

- `MEMORY_ONLY`: Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and recomputed on the fly each time they're needed. This is the default level.
- `MEMORY_ONLY_SER`: Store RDD as serialized Java objects (one-byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
- `MEMORY_AND_DISK`: Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk and read them from there when needed.

- `MEMORY_AND_DISK_SER`: Similar to `MEMORY_ONLY_SER`, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
- `DISK_ONLY`: Store the RDD partitions only on disk.

The cache will use `memory_only`. You can use persist for all the levels of storage using the `persist` function:

```
df.persist()
```

If you want to remove the data in the storage level manually, you can use the `unpersist` function:

```
df.unpersist()
```

Table 3-2. *Storage Levels in Spark*

Level	Space Used	CPU Time	In Memory	On Disk	Serialized
`MEMORY_ONLY`	High	Low	Yes	No	No
`MEMORY_ONLY_SER`	Low	High	Yes	No	Yes
`MEMORY_AND_DISK`	High	Medium	Partial	Partial	Partial
`MEMORY_AND_DISK_SER`	Low	High	Partial	Partial	Yes
`DISK_ONLY`	Low	High	No	Yes	Yes

Saving Data

After completing all the data operations, you have multiple options for saving your dataset, either as a Hive table or as a text file. Let's start by examining an example of how you can save the output DataFrame as a text file using the `write` and `save` methods. In the following command, you can specify the format of the output file along with the option of setting a delimiter. We choose a pipe delimiter for demonstration purposes, but you can customize and choose any meaningful delimiter that makes sense for your dataset.

```
df.write.format('csv').option('delimiter', '|').save('output_df')
```

Suppose you are working on a cluster and have multiple partitions of the data; you can save them together in a single file using a `coalesce` function. In that case, you would change the preceding command in the following way:

```
df.coalesce(1).write.format('csv').option('delimiter', '|').save
('output_coalesced_df')
```

If you have to partition the data by any column for easy indexing, you can do that while saving using the `partitionBy` function, as shown here:

```
df.write.partitionBy('release_year').format('csv').option
('delimiter', '|').save('output_df_partitioned')
```

If you want to save your data as a Hive table, you can use the `saveAsTable` function:

```
df.write.saveAsTable('film_ratings')
```

Pandas Support

Yes, we are working with PySpark. But there might be times when you want to convert your PySpark DataFrame to a pandas DataFrame. You can do so via the `toPandas()` function. Warning: Do not attempt to run the following command on large datasets, as you will encounter memory issues. This is because you are copying your distributed dataset to local memory. Once a pandas DataFrame is created, there is no connection between the newly created pandas DataFrame and Spark.

```
# Pandas to PySpark
df_pandas=df.toPandas()
# Pandas to PySpark
df_py = spark.createDataFrame(df_pandas)
```

Joins

Figure 3-8 shows quick pictorial representations of all joins. The following section teaches you about each join and explains how it works.

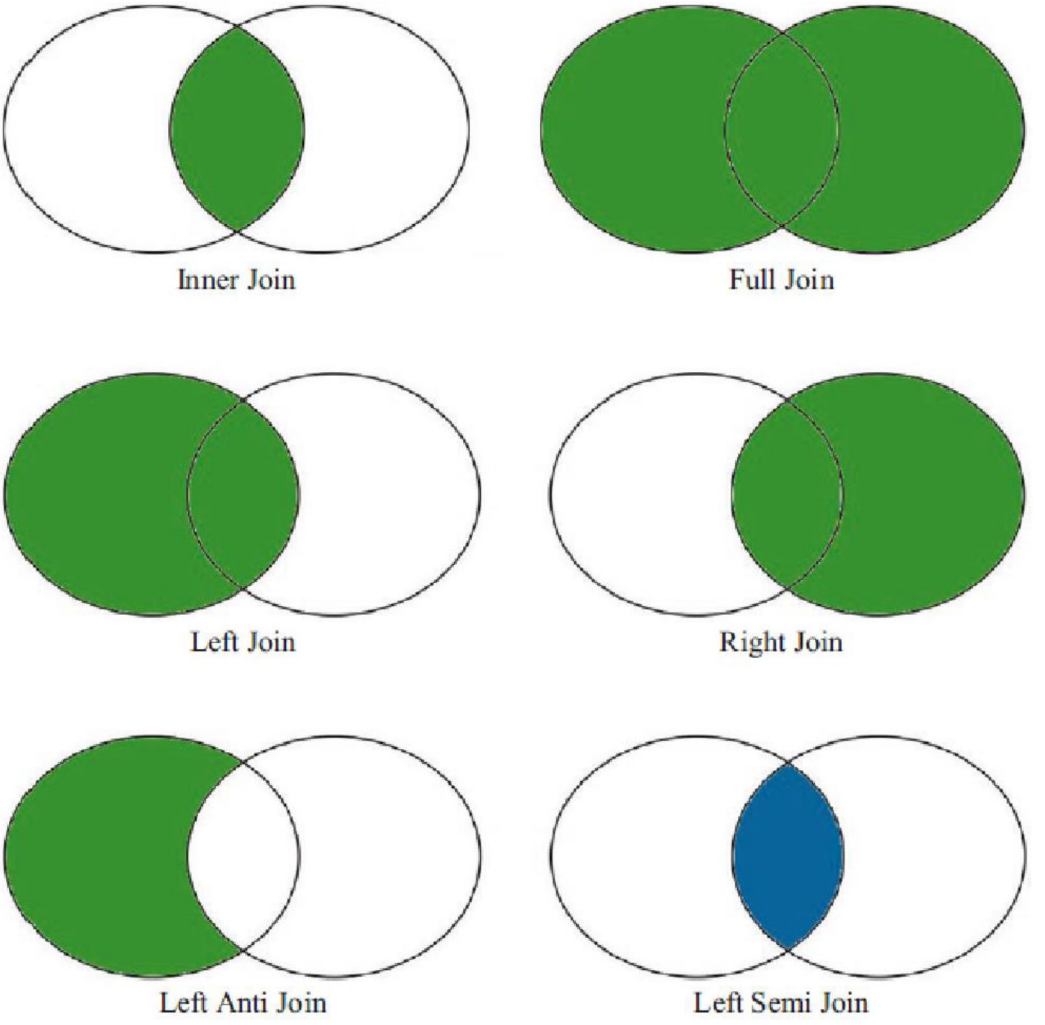


Figure 3-8. *Types of joins*

Joins are one of the most frequent operations you do with DataFrames. For joins, we introduce you to another dataset with some additional information, which we will work with in conjunction with the data we have been working with so far. See Table 3-3.

Table 3-3. *Metadata of Additional Data*

Variable	Description
id	Unique identifier for the movie
cast	List of cast members
adult	Indicator if the movie is an adult one or not
directors	Film director's information
vote_count	Total number of votes casted for popularity
spoken_languages	List of languages used in the movie
Poster_path	Path where the poster image for the movie is stored
homepage	Web page of the movie, if exists
imdb_id	IMDB ID pertaining to the movie
genres	List of genres of the movie
video	Video indicator on the website

First load this data as a new DataFrame and select all the rows with non-missing IDs, as follows:

```
# Input parameters
file_location = "movie_data_part2.csv"
file_type = "csv"
infer_schema = "False"
first_row_is_header = "True"
delimiter = "|"
# Bring all the options together to read the csv file
df_p1 = Spark.read.format(file_type) \
.option("inferSchema", infer_schema) \
.option("header", first_row_is_header) \
.option("sep", delimiter) \
.load(file_location)
# Cast identifer into the right datatype
df_p1 = df_p1.withColumn('id',df_p1['id'].cast("integer"))
# Filter missing values
df_p1=df_p1.filter((df_p1['id'].isNotNull()) & (~isnan(df_p1['id'])))
```

Right away, the common variable you will observe in both datasets is ID. We will be able to join the DataFrames using this identifier. Multiple joins are available in PySpark, so let's go through each.

- **Inner join:** This joins both DataFrames on the identifier and retains only the common rows and columns (see Figure 3-9).

```
# Inner Join
df.join(df_p1, df['id'] == df_p1['id'],'inner').printSchema()
df.join(df_p1, df['id'] == df_p1['id']).count()
```

```
root
 |-- id: integer (nullable = true)
 |-- budget: float (nullable = true)
 |-- popularity: float (nullable = true)
 |-- release_date: date (nullable = true)
 |-- revenue: float (nullable = true)
 |-- title: string (nullable = true)
 |-- mean_popularity: double (nullable = false)
 |-- varaiance: double (nullable = true)
 |-- release_year: integer (nullable = true)
 |-- cast: string (nullable = true)
 |-- adult: string (nullable = true)
 |-- directors: string (nullable = true)
 |-- vote_count: string (nullable = true)
 |-- spoken_languages: string (nullable = true)
 |-- poster_path: string (nullable = true)
 |-- homepage: string (nullable = true)
 |-- imdb_id: string (nullable = true)
 |-- genres: string (nullable = true)
 |-- video: string (nullable = true)
 |-- id: integer (nullable = true)


43783
```

Figure 3-9. *Output columns and counts of inner join*

Spark has processed the join of the DataFrames and included the identifier twice. As a good practice, you can either rename the identifier with the same name or drop the duplicate identifier altogether.

By default, Spark uses an inner join if no parameter is passed.

- **Left/Left outer join:** A left join merges both DataFrames on the identifier, retains all the rows of the left-hand DataFrame, and matches any rows common with the right-hand DataFrame. If there is no equivalent row in the right-hand DataFrame, Spark will insert a null for all the columns.

  ```
  # Left Join
  df.join(df_p1, df['id'] == df_p1['id'],'left').count()
  ```

  Output:

  ```
  43998
  ```

- **Right/Right outer join:** This merges both DataFrames on the identifier, retains all the rows of the right-hand DataFrame, and matches any rows that are common with the left-hand DataFrame. If there is no equivalent row in the left-hand DataFrame, Spark will insert a null for all the columns.

  ```
  # Right Join
  df.join(df_p1, df['id'] == df_p1['id'],'right').count()
  ```

  Output:

  ```
  43783
  ```

- **Full outer join:** This joins both DataFrames on the identifier and retains all the rows of both the left and right DataFrames that have the same identifier. If there are no equivalent rows in either of the DataFrames, Spark will insert a null for all the columns.

  # Outer Join

  ```
  df.join(df_p1, df['id'] == df_p1['id'],'outer').count()
  ```

  Output:

  ```
  43998
  ```

- **Left Anti join:** This joins both the DataFrames on the identifier and retains all the rows of left-hand DataFrame that are present in the right-hand DataFrame. It also only retains the left-hand DataFrame schema (see Figure 3-10).

```
# Left Anti Join
df.join(df_p1, df['id'] == df_p1['id'],'left_anti').
printSchema()
df.join(df_p1, df['id'] == df_p1['id'],'left_anti').count()
```

```
root
 |-- id: integer (nullable = true)
 |-- budget: float (nullable = true)
 |-- popularity: float (nullable = true)
 |-- release_date: date (nullable = true)
 |-- revenue: float (nullable = true)
 |-- title: string (nullable = true)
 |-- mean_popularity: double (nullable = false)
 |-- varaiance: double (nullable = true)
 |-- release_year: integer (nullable = true)

215
```

Figure 3-10. *Output columns and counts of left anti join*

- ***Left Semi join:*** This is similar to an inner join, except it does not yield the columns from the right-hand DataFrame (see Figure 3-11).

```
# Left Semi Join
df.join(df_p1, df['id'] == df_p1['id'],'left_semi').
printSchema()
df.join(df_p1, df['id'] == df_p1['id'],'left_semi').count()
```

```
root
 |-- id: integer (nullable = true)
 |-- budget: float (nullable = true)
 |-- popularity: float (nullable = true)
 |-- release_date: date (nullable = true)
 |-- revenue: float (nullable = true)
 |-- title: string (nullable = true)
 |-- mean_popularity: double (nullable = false)
 |-- varaiance: double (nullable = true)
 |-- release_year: integer (nullable = true)



43783
```

Figure 3-11. *Output columns and counts of left anti join*

- **Broadcasting:** There will be situations where you have to join a large dataset (millions of rows) and a small dataset (hundreds of rows). To join them efficiently, Spark provides another function called `broadcast`. The smaller DataFrame will be replicated in all the nodes of the cluster. This improves efficiency, as Spark doesn't need to do any shuffling in the larger dataset. Here is a working example:

  ```
  # Broadcast Join
  df.join(broadcast(df_p1), df['id'] == df_p1['id'],
  'left_semi').count()
  ```

  Output:

  ```
  43998
  ```

Dropping Duplicates

When working with real-world data, another handy function that you may end up using is `dropDuplicates`. You can drop duplicates of a single row or multiple rows. Say you want to drop all the duplicates in your DataFrames where the title and year are the same; you can do so using the following command:

```
# Dropping Duplicates
df.dropDuplicates(['title','release_year']).count()
```

Output:

```
43643
```

```
# Original Data count
df.count()
```

Output:

```
43998
```

Data Visualizations

Well, the sad news is that Spark has no built-in data visualization packages. Then what are we discussing here? The Python has a rich library of data visualization packages such as `matplotlib` and `seaborn`. You can use these packages in concert with the DataFrames and create the required plots.

If you take a step back, you can see that you use aggregated data mostly for visualization. If you have to understand a continuous variable, the best plot you can use is a histogram. Similarly, for categorical data ,you want to see the group plot. So, you can use the power of Spark to calculate these aggregations and then top it off with the nice visualization packages that are available in Python.

First, let's plot the histogram of the popularity data (see Figure 3-12):

```
# Step 1: Importing the required libraries
%matplotlib inline
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns
# Step 2: Processing the data in Spark. We can use the histogram function
from the RDD
histogram_data = df.select('popularity').rdd.flatMap(lambda x:
x).histogram(25)
# Step 3: Loading the Computed Histogram into a pandas DataFrame for
plotting
hist_df=pd.DataFrame(list(zip(*histogram_data)), columns=['bin',
'frequency'])
```

```
# Step 4: Plotting the data
sns.set(rc={"figure.figsize": (12, 8)})
sns.barplot(hist_df['bin'],hist_df['frequency'])
plt.xticks(rotation=45)
plt.show()
```

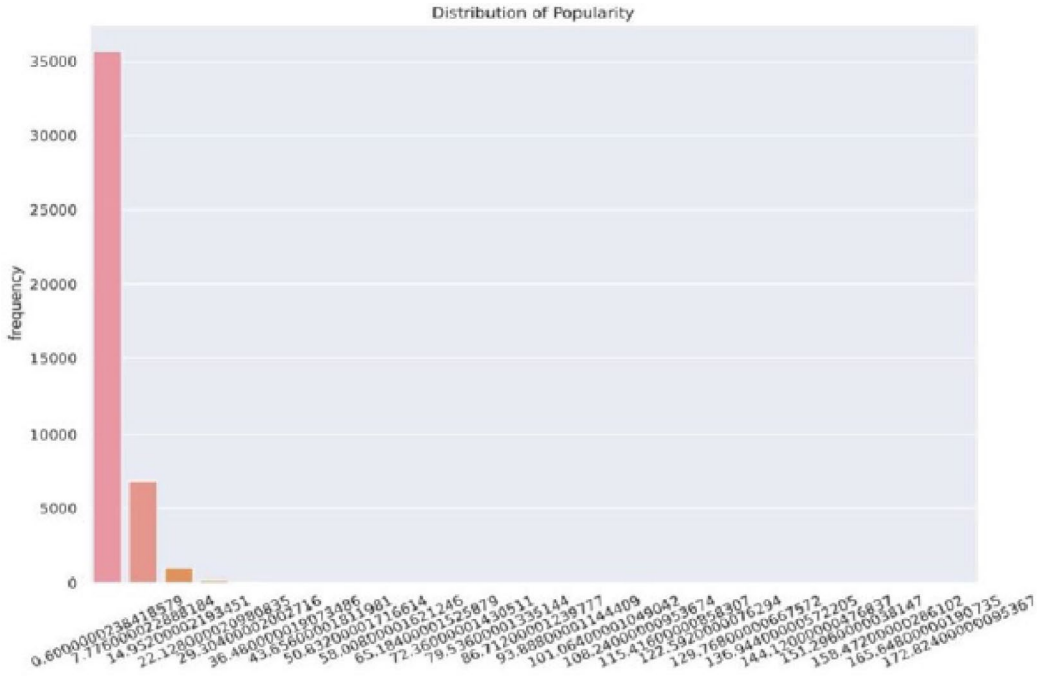


Figure 3-12. *Histogram of popularity*

Let's break down what is happening in each step. First, we imported the required libraries. We are doing all the heavy lifting pertaining to the data in Step 2. We are using the `histogram` function available in the RDD API to discretize the distribution and calculate the bins. In this procedure, we requested that it discretize our data into 25 buckets. In Step 3, we are creating a pandas DataFrame on top of the data and then using it for plotting. Here's the hack. We are not directly using the histogram functions in Python; all the calculations are done in Spark. We use a bar plot on top of the histogram data to look at the distribution. As you can observe, the distribution is hard to interpret because of the sparseness of the values in the tail.

Let's try filtering the data to see if we can get a better idea of the distribution.

```
# Filtering the data to get a better understanding of data
df_fil=df.filter('popularity<22')
# Processing the data in Spark. We can use the histogram function
from the RDD
histogram_data = df_fil.select('popularity').rdd.flatMap(lambda x: x).
histogram(25)
# Loading the computed histogram into a pandas DataFrame for plotting
hist_df=pd.DataFrame(list(zip(*histogram_data)),
columns=['bin', 'frequency'])
# Plotting the data
sns.set(rc={'figure.figsize':(11.7,8.27)})
sns.barplot(hist_df['bin'],hist_df['frequency'])
plt.xticks(rotation=25)
plt.title('Distribution of Popularity - Data is filtered')
plt.show()
```

We repeat the same steps except for filtering. Now we get a much better distribution (see Figure 3-13).

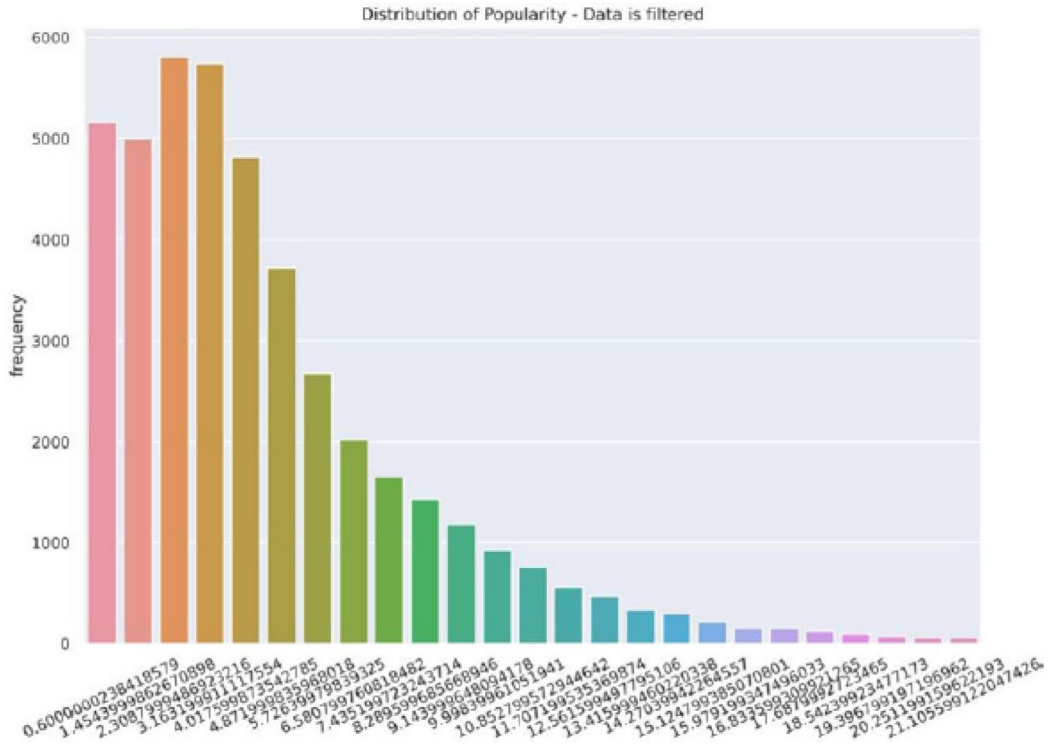


Figure 3-13. *Histogram of popularity after filtering*

Let's try to plot another variable. Say you want to know how many films were released between 1960 and 1970 by the year in the dataset. You can determine this by using the following commands (see Figure 3-14):

```
# Step 1: Preparing the data using Spark functions and converting to pandas
DataFrame
df_cat=df.filter("(release_year>1959) and (release_year<1971)").
groupby('release_year').count().toPandas()
# Step 2: Sorting the values for display
df_cat=df_cat.sort_values(by=['release_year'], ascending=False)
# Step 3: Plotting the data
sns.set(rc={'figure.figsize':(11.7,8.27)})
sns.barplot(df_cat['release_year'],df_cat['count'])
plt.xticks(rotation=25)
plt.title('Number of films released each year from 1960 to 1970 in our dataset')
plt.show()
```

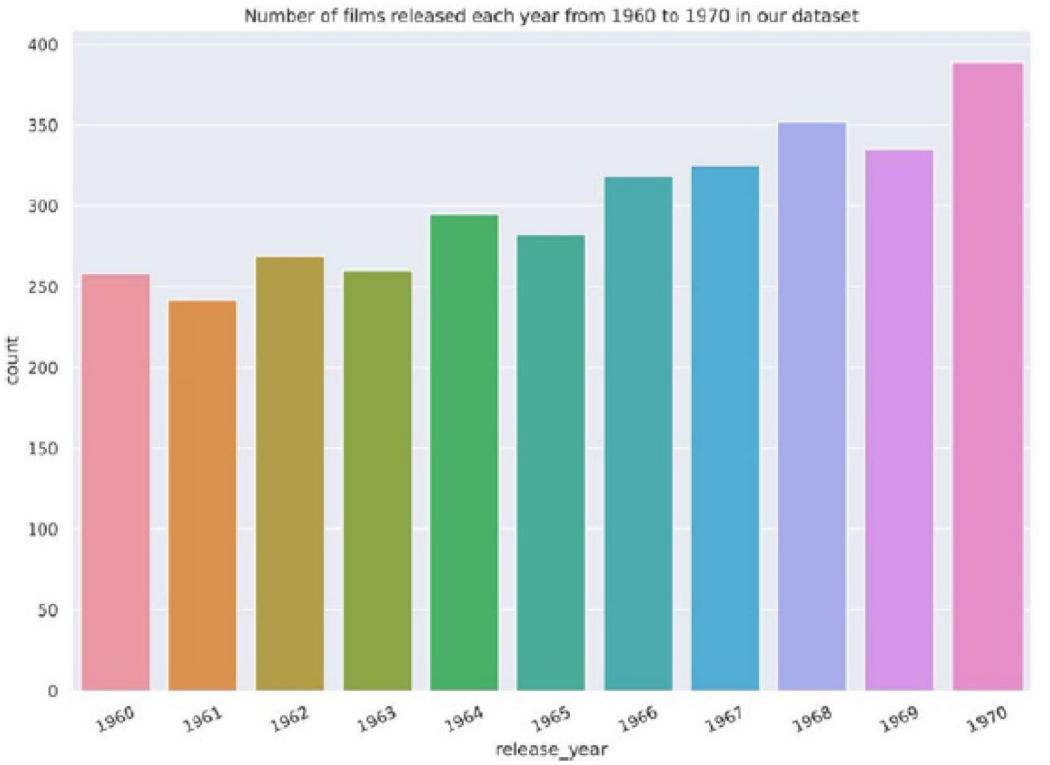


Figure 3-14. *Number of films released every year from 1960 to 1970*

Use the following dataset for Exercise 3-1: https://www.kaggle.com/kakarlaramcharan/tmdb-data-0920.

EXERCISE 3-1: DATA MANIPULATIONS

Question 1: Identify the second most popular movie in 2010 based on popularity.

Question 2: Identify all title names that are repeated and show the years in which they were repeated.

Question 3: Identify the top movies by popularity across all years.

Question 4: Visualize the budget of movies in 2014 and 2015.

Introduction to Machine Learning

Before you jump into the world of machine learning, you need to understand the progression of analytics. Analytics is a computational analysis of data or statistics. It is used for the discovery, interpretation, and communication of meaningful patterns of data. You might hear multiple terms and buzzwords related to analytics, but they are all used for collecting insights and making informed decisions.

Analytics can broadly be divided into the following (see Figure 3-15):

Descriptive: This field of analytics is focused on creating reports and charts understanding the data.

Predictive: This field of analytics is associated with estimating what will happen given what has happened. It relies on historical data to make predictions.

Prescriptive: This field of analytics is focused on what has to be done given past data. This includes decision optimization.

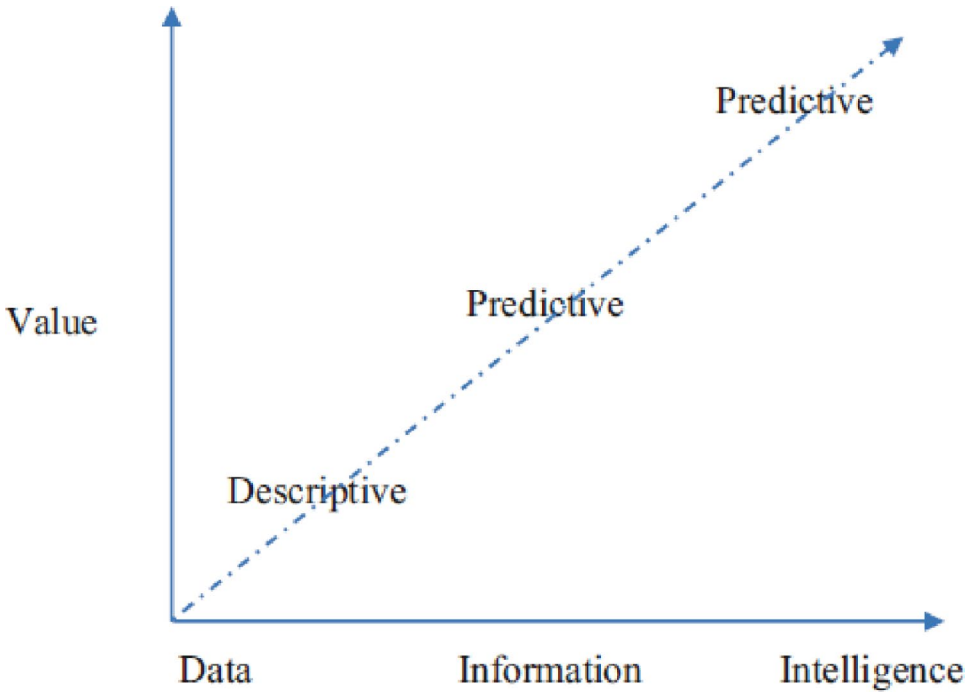


Figure 3-15. *Overview of analytics*

Often these terms are used in a way that can confuse new learners. Sure, there is an overlap. Machine learning is a subfield of computer science and is considered a subset of artificial intelligence. *Predictive analytics* refers to statistical modeling and is a term commonly found in the insurance, banking, and marketing industries. *Machine learning* is a broader term that encompasses multiple categories, including the following:

> **Supervised learning:** In this type of learning, an algorithm is provided a historical dataset with target labels or events. This field of study is validated through measures of accuracy. This also includes any time-series modeling.
>
> **Unsupervised learning:** In this type of learning, an algorithm is provided a historical dataset with no target labels or events. The intent here is to identify the hidden patterns or segments in the data.

You need to understand the data types before we discuss the types of learning techniques found in the supervised and unsupervised methods. We can broadly classify data into qualitative and quantitative.

Qualitative data is something that can't be measured. This can include the color of your vehicle, marital status, education, and so on. These variables can also be called categorical variables. We can further classify qualitative data into the following:

- **Nominal** – When there are multiple levels in the data and no natural ordering, you can consider it to be nominal. For example, the color of your vehicle, or your marital status.
- **Ordinal** – When there is an inherent ordering in your data, you can consider it to be ordinal. For example, education. In education you can have multiple levels, such as less than high school, high school, graduate, and postgraduate. These levels have a ranking associated with them. We know postgraduate is on the higher end of the spectrum, while less than high school is on the lower end of the spectrum.

Quantitative data can be measured. This can include age, distance, and so on. Quantitative data can be further divided into discrete and continuous:

- **Discrete** – You can consider data to be discrete when it has integer values; for example, age (10, 20).
- `Continuous` – You can consider data to be continuous when it has fractional or decimal data; for example, distance (40.71 miles, 59.82 miles).

We have dedicated a chapter to unsupervised learning/clustering in which you will learn these different techniques and methods (see Figure 3-16).

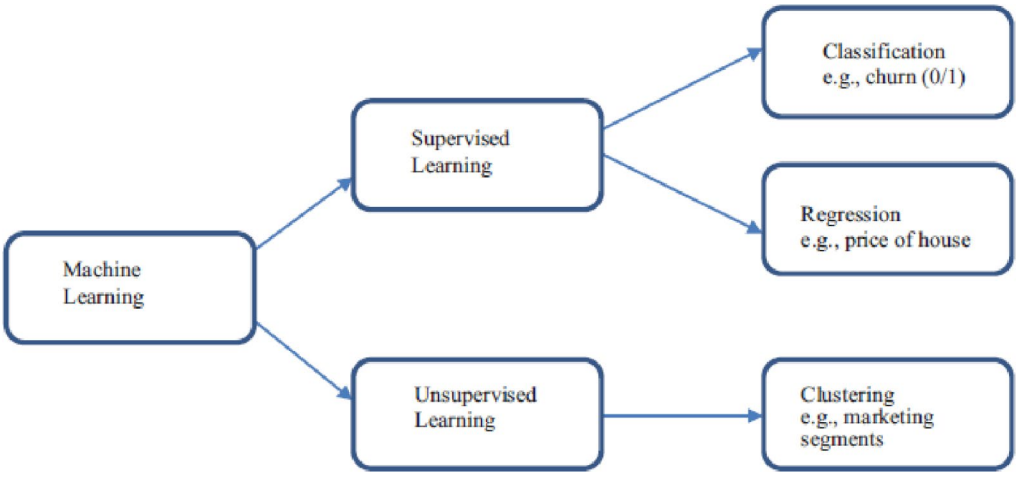


Figure 3-16. *Classification of machine learning*

Why is it important to know these datatypes? Well, because it will help you determine what types of algorithms can be applied. Certain algorithms are designed for selected datatypes. In supervised learning, there are two different classes of algorithms: classification and regression:

> **Classification:** In this class of machine learning, the target/event data is discrete. Examples include predicting the customers who are likely to churn (0/1) or the customers who are likely to default on a loan (0/1).

Regression: In this type of machine learning, the target/event data is continuous. Examples include predicting the price of a house ($) based on given parameters or predicting customers' purchases in a given month.

In later chapters, we explore which algorithms can be applied to each of these methods.

Clustering can be defined as grouping homogeneous groups of data in a dataset.

There are standard methodologies most data professionals follow for predictive modeling. One is called CRISP–DM, or Cross-Industry Standard for Data Mining (see Figure 3-17). It consists of six major milestones, as follows:

1. **Business Understanding:** This is the first important step and involves understanding the problem statement and formulating it into a data problem. This comes from the business.
2. **Data Understanding:** This step includes data collection and descriptive data analysis to discover the data and become familiar with interesting features.

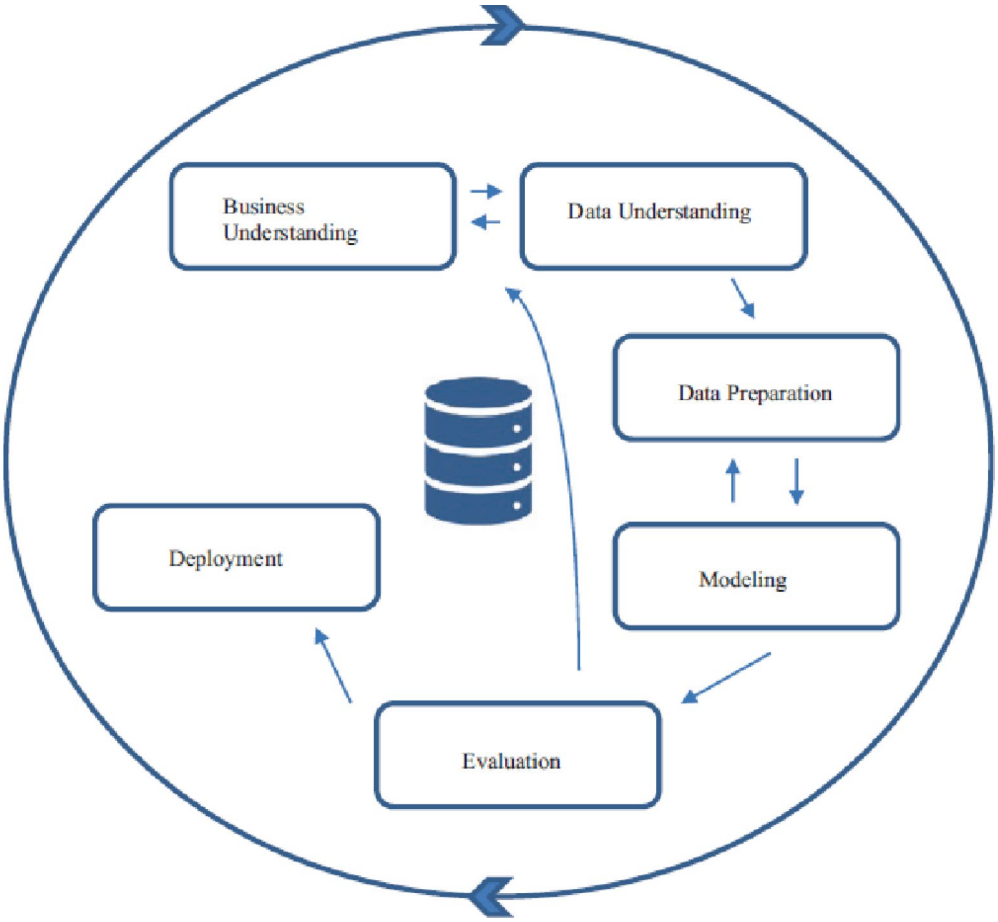


Figure 3-17. *CRISP–DM methodology*

3. **Data Preparation:** This includes consolidating all the data features and creating additional features/variables from the dataset.
4. **Modeling:** This includes applying appropriate algorithms and iterating to create more variables if necessary.

5. **Evaluation:** Once one or multiple models are built, a model is selected based on the validation criteria set. We have a separate chapter to help you understand all the model evaluation and validation techniques. We also seek business feedback on whether the model is achieving the intended function.
6. **Deployment:** This involves putting the model into production and making the model scores available for various stakeholders across the business.

An alternate standard framework that is also common among data practitioners is SEMMA. SEMMA stands for Sample, Explore, Model, Modify, and Assess. This methodology was introduced by SAS prior to CRISP–DM. This modeling-focused approach does not include business inputs as part of its process. It works well in decentralized organizations if you clearly define your business objective.

Summary

- You explored the possibilities of creating window functions and pondered specific business questions.
- You learned sampling, how it can be implemented, and why it is important.
- You learned the mechanics of the `cache` and `persist` functions and when they should be used.
- You learned simple hacks for visualizations.
- You are now equipped with machine learning basics and the framework.

Good job! The next chapter introduces variable selection and discusses different techniques. When you are inundated with data, you can always count on the variable selection chapter. It will help you identify the top variables with information. See you in the next chapter!

CHAPTER 4

Variable Selection

Variable selection is an art. This chapter aims to help you understand the different variable selection techniques that can be used to select the best features in your dataset. It is one of the critical processes in data science. To put it simply, say you are the coach of a soccer team. You want to pick the best team to win the World Cup. You need the best player in each position (*best features*), and you don't want too many players who play the same position (*multicollinearity*) at a given time. In PySpark, there are multiple ways to accomplish this task. This chapter goes through some built-in and custom-built variable selection techniques (see Figure 4-1).

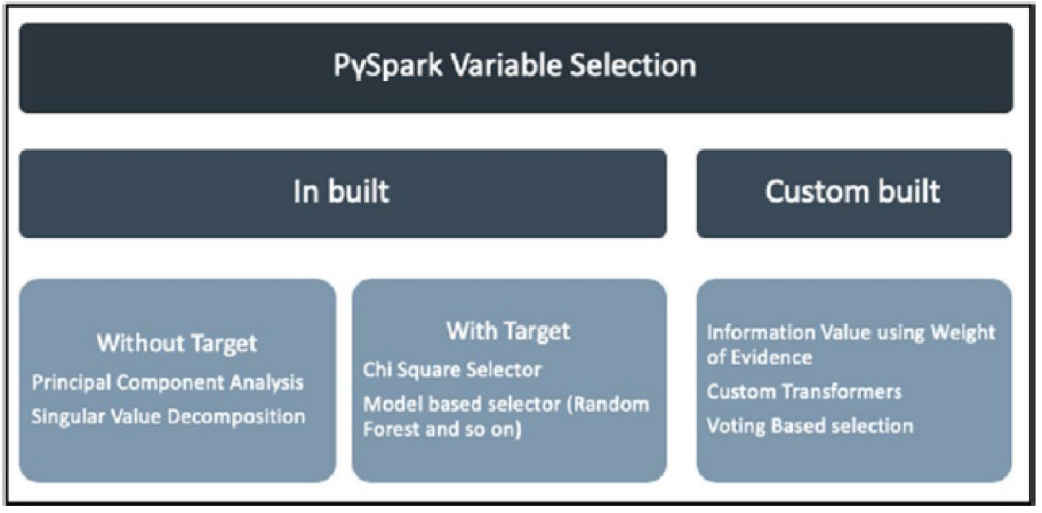


Figure 4-1. *Variable selection*

The upcoming pages explore the concepts behind each technique, do some calculations by hand to see how they work *(whenever necessary)*, and finally implement them using PySpark. Throughout this chapter, we use the bank dataset sourced from the

© Ramcharan Kakarla, Sundar Krishnan, Balaji Dhamodharan and Venkata Gunnu 2024
R. Kakarla et al., *Applied Data Science Using PySpark*, https://doi.org/10.1007/979-8-8688-0820-3_4

UCI Machine Learning Repository at https://archive.ics.uci.edu/ml/datasets/Bank+Marketing. You need to download the bank.zip file from the data folder to access the bank-full.csv file. Before you proceed, consider this intuitive way to look at variable selection in Figure 4-2.

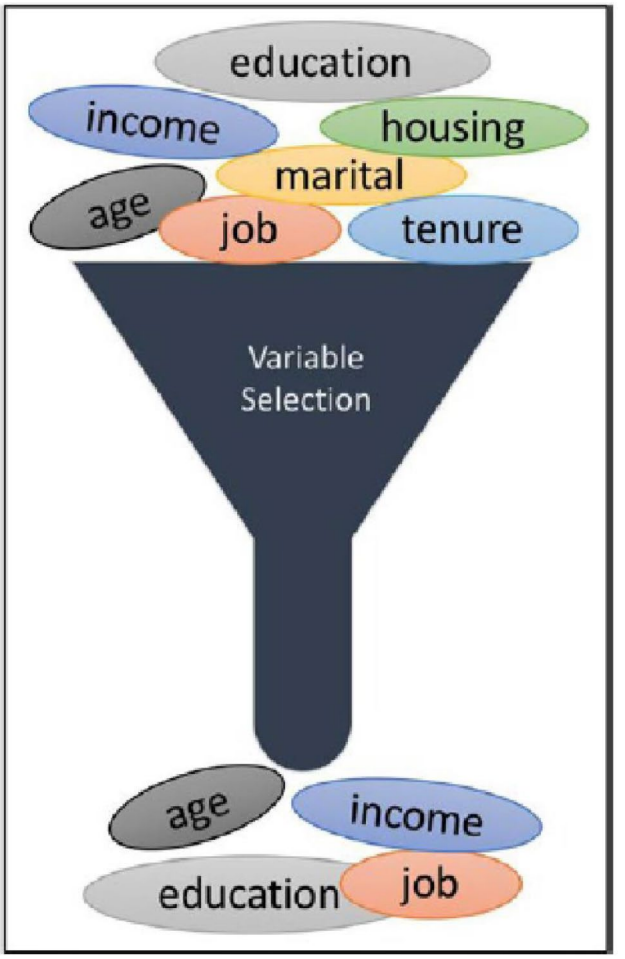


Figure 4-2. *Intuitive way to think about variable selection*

In Figure 4-2, many features are initially considered (shown on the top of the funnel). However, only the required and critical features are finally selected using the variable selection techniques (the bottom of the funnel). By selecting the right set of features (removing redundant or irrelevant variables), model accuracy and performance can be improved, thus mitigating the curse of dimensionality. So variable selection is an iterative and key step in the data science process.

Exploratory Data Analysis

Exploratory data analysis (EDA) is used to analyze and investigate datasets to identify characteristics and patterns using summarization or visualization methods. The previous chapters extensively covered this topic with numerous code examples. We recommend you apply these code snippets to any dataset you work with. When you understand the data better, you can make quick decisions and explain the reasoning behind the decisions. EDA can also become a powerful first step in the variable selection process. If you are wondering how to do this, consider cardinality (see the definition in this section) and missing values. It is good practice to check these two things when doing any machine learning exercise.

Let's use this example of home prices to better understand cardinality and missing values (see Figure 4-3).

Zip	Home type	HOA fees	Sq. Ft	Price
19102	Town Home	$100	2400	$350K
19103	Condo	$300	1200	$200K
19145	Single Family Unit	NaN	2500	$400K
19150	Single Family Unit	NaN	1800	NaN
19040	Town Home	$250	2275	$325K

Figure 4-3. *Home prices dataset*

Cardinality

Cardinality is the number of unique values of a variable. Assume that you are building a model to predict house prices. The `Home type` variable is "Single Family Unit" for all the records. In this case, the cardinality for this variable is 1 ("Single Family Unit"). *Ho*me `type` is not contributing any useful information because, for any record you choose, this variable provides the same information ("Single Family Unit"). Therefore, you can omit this variable and use other variables to build the predictive model. Although you can calculate the cardinality of any variable, cardinality is more useful for categorical variables. In Figure 4-3, the cardinality of `Home type` is 3, indicating there are three distinct values ("Town Home", "Condo", and "Single Family Unit").

Missing Values

As the name suggests, missing values refer to missing pieces of information. You need to understand the reason behind the missing piece of information to build a better model. In general, data can be missing for the following three reasons:

- Missing At Random (MAR)
- Missing Completely At Random (MCAR)
- Missing Not At Random (MNAR)

By understanding the type of missing information, you can treat the data accordingly. The next section uses the same home price example to explain missing values.

Missing at Random (MAR)

When a piece of data is missing at random, there is a relationship between the missing data and the observed data. The home price example can help explain this. When you look closely at the `Home type` and `HOA fees` (Homeowners' Association fees) variables, you can see that there is a value for `HOA fees` when `Home type` is "Town Home" or "Condo" and the fee is missing when the `Home type` is "Single Family Unit." This makes sense because a single family unit (in many cases) does not have HOA fees. In this case, `HOA fees` for "Single Family Unit" homes can be imputed with zero. This type of missing data is called *missing at random (MAR).*

Note *Imputing* is a way of filling in missing values in data science literature.

Missing Completely at Random (MCAR)

When a piece of data is missing completely at random, there is no relationship between the missing data and the observed data. Consider the `Price` variable in the data. One of the home prices is missing. Unlike the MAR example , there is no logic that you can derive from this missing data. There could be a lot of reasons behind the missing information. Maybe the house is listed for auction, or the house is a new construction, and the seller is yet to determine a price, or there could be some other reason. In any case, the missing price has nothing to do with the observed data. This is a typical

example for data *missing completely at random (MCAR)*. To impute this data, you might have to use your logic. We do have some recommendations for how to treat this type of missing data, as follows:

- Mean, median, or mode imputation
- Model-based imputation (this is discussed in future chapters)
- Multiple imputation
- Use business logic to impute the missing information
- Drop features with significant missing data (variable selection)
- Drop rows with missing values (least recommended due to potential information loss)

How do you use business logic to impute the missing information? Let's assume you work for Zillow and know that the `Price` information is missing only when the house is listed for auction. In this case, your team recommends you use a default price for houses listed for auction (say $1). Then, you would impute the price with $1.

Missing Not at Random (MNAR)

When data is missing not at random, it means that the data was not collected. In the current example, we do not know when the house was built (*year*). This information is missing because we did not account for the `Year` variable in our data collection process. To resolve this issue, you should understand and improve the data collection process to account for the missing information.

We have discussed the types of missing values in detail. Can you use this to reject variables in the dataset? Yes, you can. Most data science folks reject a variable based on the percentage of the data missing. For example, you can reject a variable with over 80 percent missing values. The 80 percent is not a standard value and can vary on a case-to-case basis.

EXERCISE 4-1: MISSING VALUES AND CARDINALITY

Download the Melbourne Housing dataset: https://www.kaggle.com/dansbecker/melbourne-housing-snapshot/data#.

Question 1: Calculate the cardinality for each variable in the dataset. Can you reject any variables based on cardinality? (Hint: Look for cardinality of 1.)

Question 2: Calculate the missing value percentage for each variable in the dataset. Can you eliminate any variables based on missing values? (Hint: Just for this exercise, reject any variable that has more than 45 percent missing values.)

Question 3: Impute the `YearBuilt` column with a median value. (Hint: Use 0.1 for `relativeError`.)

Question 4: Impute the `BuildingArea` column to create a new variable called `mean_imputed_BuildingArea` and calculate the new mean after imputation.

Returning to the bank dataset, let's load the dataset and perform EDA-based variable selections.

```
filename = "bank-full.csv"
target_variable_name = "y" #This variable can be set later too
```

Load the data using the following code:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
df = spark.read.csv(filename, header=True, inferSchema=True, sep=';')
df.show()
```

You need to provide a semicolon (`;`) as the separator for this dataset or it won't work. In addition, note the new option, `inferSchema`, in the file load operation. When enabled, this option automatically infers the datatype of each column in the dataset. Figure 4-4 shows the schema inference with and without this option.

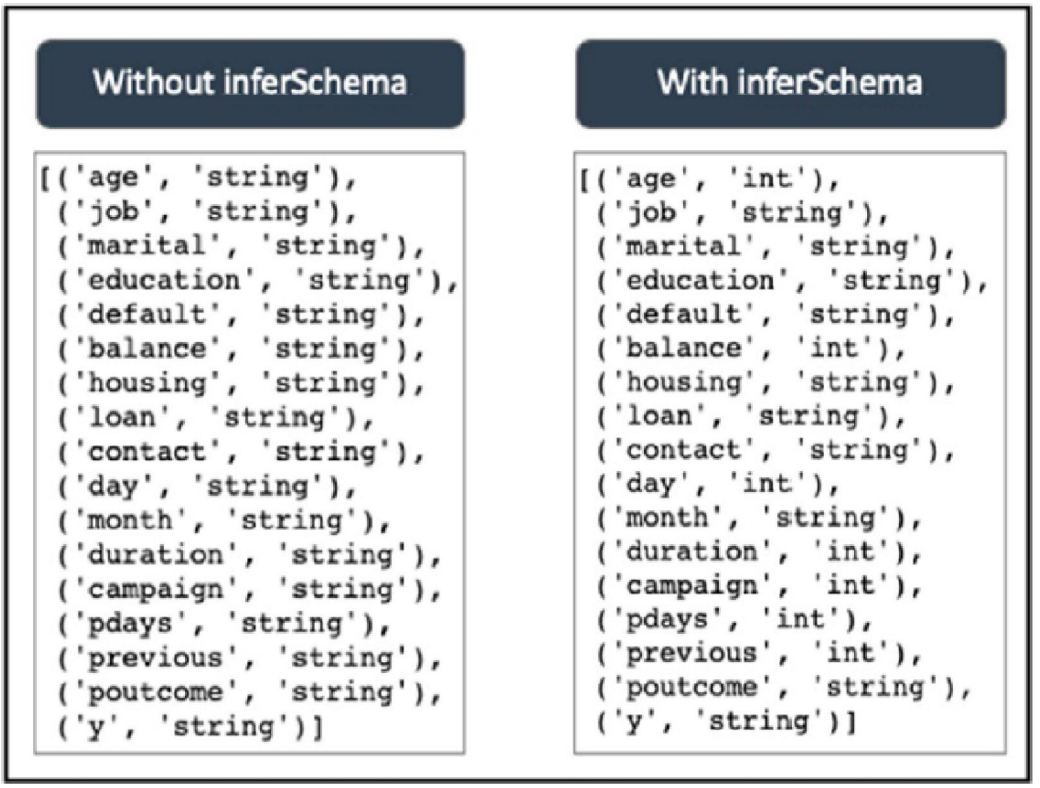


Figure 4-4. *inferSchema option output difference*

Once you have loaded the data into your environment, consider this code to get you started playing with the data:

```
#Length of the data
df.count() #45211
#describes data
df.describe().toPandas() #missing and cardinality followup
#type of each variable
df.dtypes
df.printSchema()
#single variable count by group
df.groupBy('education').count().show()
#target count
df.groupBy(target_variable_name).count().show()
#multiple column group by
```

```
df.groupBy(['education',target_variable_name]).count().show()
#column aggregations
from pyspark.sql.functions import *
df.groupBy(target_variable_name).agg({'balance':'avg', 'age':
'avg'}).show()
```

We want to highlight the described data output in Figure 4-5 and connect it to the cardinality and missing values discussed earlier. The `count` row in the `summary` column gives the count of non-missing values in a variable. The difference between the total observations and this column gives you the total number of missing values in each column. Let's review two more stats here: the `min` and `max` rows in the `summary` column. When the `min` and `max` rows have the same value for a variable, the cardinality of the variable is 1. If you are unable to connect, no worries. You can use the following code to perform these checks.

	summary	age	job	marital	education	default
0	count	45211	45211	45211	45211	45211
1	mean	40.93621021432837	None	None	None	None
2	stddev	10.618762040975401	None	None	None	None
3	min	18	admin.	divorced	primary	no
4	max	95	unknown	single	unknown	yes

Figure 4-5. *Column aggregation summary*

Code 1: Cardinality Check

This code snippet allows you to quickly analyze the cardinality (number of unique values) of each column in a Spark DataFrame, identify any columns with low cardinality based on a specified threshold (`cut_off`), and retrieve the results in a structured format using pandas for easier manipulation and interpretation.

```
# Cardinality Check
from pyspark.sql.functions import approxCountDistinct, countDistinct
"""
Note: approxCountDistinct and countDistinct can be used interchangeably.
Only difference is the computation time.
"approxCountDistinct" is useful for large datasets
"countDistinct" for small and medium datasets.
"""
def cardinality_calculation(df, cut_off=1):
    cardinality = df.select(*[approxCountDistinct(c).alias(c) for c in
    df.columns])
    ## convert to pandas for efficient calculations
    final_cardinality_df = cardinality.toPandas().transpose()
    final_cardinality_df.reset_index(inplace=True)
    final_cardinality_df.rename(columns={0:'Cardinality'}, inplace=True)
    #select variables with cardinality of 1
    vars_selected = final_cardinality_df['index']
[final_cardinality_df['Cardinality'] <= cut_off]
    return final_cardinality_df, vars_selected
cardinality_df, cardinality_vars_selected = cardinality_calculation(df)
```

Code 2: Missing Values Check

```
#missing values check
from pyspark.sql.functions import count, when, isnan, col
# miss_percentage is set to 80% as discussed in the book
def missing_calculation(df, miss_percentage=0.80):
    #checks for both NaN and null values
    missing =
df.select(*[count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in
df.columns])
# Calculate total length of the DataFrame
    length_df = df.count()
    ## convert to pandas for efficient calculations
    final_missing_df = missing.toPandas().transpose()
    final_missing_df.reset_index(inplace=True)
```

```
    final_missing_df.rename(columns={0:'missing_count'}, inplace=True)
    final_missing_df['missing_percentage'] =
final_missing_df['missing_count']/length_df
    #select variables with cardinality of 1

vars_selected = final_missing_df['index'][final_missing_df['missing_
percentage'] >= miss_percentage]
    return final_missing_df, vars_selected
missing_df, missing_vars_selected = missing_calculation(df)
```

For the bank dataset, no columns are rejected based on these two checks. However, in real-world datasets, you will see a handful of columns that can be eliminated after performing cardinality and missing value checks. That is all you need to learn for EDA-based variable selection techniques. Before we move ahead with other selection techniques, we go through one more feature-engineering exercise in EDA. This will ensure that your data is ready for downstream tasks. Remember Figure 4-4, where we still have some string columns. The next set of selection techniques requires all the columns to be numeric (integers or floats).

Converting strings to numerics can be achieved using one of the following options:

1. Individually casting columns to a desired datatype. This is cumbersome for large datasets.

2. `OneHotEncoder` (for Spark >= 3.0) or `OneHotEncoderEstimator` (for Spark >= 2.3) from `pyspark.ml.feature`. This option will create dummy variables for each category, thereby increasing the number of features you work with. In this dataset, the `education` variable has four categories: primary, secondary, tertiary, and unknown. After using `OneHotEncoder` on this column, the output looks like Figure 4-6. *Note: PySpark, by default, does not produce output like Figure 4-6; it requires some tweaking using UDF functions discussed in the previous chapter.*

Education	Education Primary	Education Secondary	Education Tertiary	Education Unkown
primary	1	0	0	0
secondary	0	1	0	0
tertiary	0	0	1	0
Unknown	0	0	0	1
Primary	1	0	0	0
secondary	0	1	0	0

Figure 4-6. *One hot encoder*

3. `StringIndexer` from `pyspark.ml.feature`. This option assigns a number to each level. The most frequent value would get index 0, followed by the next most frequent, and so on. The output for the `education` variable using `StringIndexer` would look like Figure 4-7. Although the output of `StringIndexer` looks like an ordinal output (increasing order), it should not be considered as ordinal. Most algorithms fail to treat this variable as nominal output (mutually exclusive levels). This is one of the drawbacks of `StringIndexer`.

Education	Education Index
primary	0
secondary	1
tertiary	2
Unknown	3
Primary	0
secondary	1

Figure 4-7. *StringIndexer output*

```
{'numeric': [{'idx': 0, 'name': 'age'},
  {'idx': 1, 'name': 'balance'},
  {'idx': 2, 'name': 'day'},
  {'idx': 3, 'name': 'duration'},
  {'idx': 4, 'name': 'campaign'},
  {'idx': 5, 'name': 'pdays'},
  {'idx': 6, 'name': 'previous'},
  {'idx': 7, 'name': 'job'},
  {'idx': 8, 'name': 'marital'},
  {'idx': 9, 'name': 'education'},
  {'idx': 10, 'name': 'default'},
  {'idx': 11, 'name': 'housing'},
  {'idx': 12, 'name': 'loan'},
  {'idx': 13, 'name': 'contact'},
  {'idx': 14, 'name': 'month'},
  {'idx': 15, 'name': 'poutcome'}]}
```

Figure 4-8. *Features column schema*

4. The last option is to use a weighted index, something like a weight of evidence (WOE), which is discussed later in this chapter. It overcomes the drawbacks faced by `StringIndexer` and `OneHotEncoder`. It uses the distribution of target versus the non-target population to create the weights for each category.

This example uses `StringIndexer` to convert all the columns. The code is provided next.

Step 1: Identify Variable Types

```
# Identify variable types
def variable_type(df):
    # use the dtypes to separate character and numeric variables
    vars_list = df.dtypes
    char_vars = [] #character variables list
    num_vars = [] #numeric variable list
    for i in vars_list:
```

```
        if i[1] in ('string'):
            char_vars.append(i[0])
        else:
            num_vars.append(i[0])
    return char_vars, num_vars
#apply variable_type function on our DataFrame
char_vars, num_vars = variable_type(df)
```

Step 2: Apply StringIndexer to Character Columns

Apply the `StringIndexer` function to the character variables identified in Step 1 (`char_vars`). The code is provided here:

```
from pyspark.ml.feature import StringIndexer
from pyspark.ml import Pipeline
#converts each category column to index
def category_to_index(df, char_vars):
    char_df = df.select(char_vars)
    indexers =
[StringIndexer(inputCol=c, outputCol=c+"_index", handleInvalid="keep") for
c in char_df.columns]
    pipeline = Pipeline(stages=indexers)
    char_labels = pipeline.fit(char_df)
    df = char_labels.transform(df)
    return df, char_labels
#apply category_to_index function on our DataFrame
df, char_labels = category_to_index(df, char_vars)
```

A couple of things are highlighted in this code. The first is the `StringIndexer` operation, which takes in an input column (`Education`) and produces an output column (`Education_index`). One thing to note is the `handleInvalid` option, which is set to `keep` *so* that it will still work when you encounter new data in the future. The next option is the `Pipeline` option, which executes the steps sequentially. We cover pipelines in detail in later chapters. The DataFrame output currently has the old string columns and new numeric columns. Now subset the data to the required columns as follows:

```
df = df.select([c for c in df.columns if c not in char_vars])
```

Finally, the DataFrame is now completely numeric. Folks who are interested in renaming the _index variables to the original name can use the following code to perform the operation:

```
#rename _index columns to original variable name
def rename_columns(df, char_vars):
    mapping = dict(zip([i + '_index' for i in char_vars], char_vars))
    df = df.select([col(c).alias(mapping.get(c, c)) for c in df.columns])
    return df
# apply rename_columns to our DataFrame
df = rename_columns(df, char_vars)
```

Step 3: Apply StringIndexer to Target Column

Apply the StringIndexer method to the target variable y. The method creates an output column y_index. Since y and y_index are duplicate columns, we provide additional code to drop the duplicate column and rename y_index to the original column name, y. The code is provided here:

```
# StringIndexer Initialization
indexer = StringIndexer(inputCol='y', outputCol='y_index',
handleInvalid='skip')
indexerModel = indexer.fit(df)

# Transform the DataFrame using the fitted StringIndexer model
df = indexerModel.transform(df)

# drop existing "y" column so we can rename new "y_index" column to "y"
df = df.drop(df.y)

# rename new "y_index" column to "y"
df = df.withColumnRenamed('y_index', 'y')
```

Step 4: Assemble Features

The last step is to assemble the individual variables into a single feature vector. This is useful because you can point to one variable instead of providing individual variables in the next steps. In addition, you can optionally scale the DataFrame using a StandardScaler or MinMaxScaler. This is accomplished using the following code:

```
#assemble features into one vector
from pyspark.ml.feature import VectorAssembler
#assemble individual columns to one column - 'features'
def assemble_vectors(df, features_list, target_variable_name):
    stages = []
    #assemble vectors

    assembler = VectorAssembler(inputCols=features_list,
    outputCol='features')
    stages = [assembler]
    #select all the columns + target + newly created 'features' column
    selectedCols = [target_variable_name, 'features'] + features_list
    #use pipeline to process sequentially
    pipeline = Pipeline(stages=stages)
    #assembler model
    assembleModel = pipeline.fit(df)
    #apply assembler model on data
    df = assembleModel.transform(df).select(selectedCols)
    return df

# exclude target variable and select all other feature vectors
features_list = df.columns
features_list.remove(target_variable_name)
# apply the function on our DataFrame
df = assemble_vectors(df, features_list, target_variable_name)
```

If you prefer to look at the schema of the combined vector features, you can use the following code:

```
df.schema["features"].metadata["ml_attr"]["attrs"]
```

To make them nice pandas DataFrames, you can use the following code:

```
import pandas as pd
features_df = pd.DataFrame()
# Iterate over metadata attributes in the 'features' column
for k, v in df.schema["features"].metadata["ml_attr"]["attrs"].items():
```

```
# Convert each attribute dict  'v' into Pandas dataframe and append to
feature_df    final = pd.DataFrame(v)
    features_df=features_df.append(final,ignore_index=True)
```

That's it. You now have a DataFrame (`df`) that is compatible with the downstream tasks. One thing to notice is the target variable `y` looks like Figure 4-9 after numeric conversion. This is due to the fact that the `no` option had more frequency than the `yes` option. You need to remember this so that the interpretation of the results makes sense. You can also explicitly convert the target to the desired numerical mapping based on personal choice.

```
+---+-----+
|  y|count|
+---+-----+
|0.0|39922|
|1.0| 5289|
+---+-----+
```

Figure 4-9. *Final target mapping*

Quick Exercise Try using `StandardScaler` with `VectorAssembler` to create a scaled DataFrame in the previous step. (Hint: You need to add the scaler model in the stages, along with assembler.)

Built-in Variable Selection Process: Without Target

In a machine learning model setting, you can reduce the dimension of the data without an explicit target. This type of dimensionality reduction is called *data compression*. Think of an image file. You have multiple ways to compress the file, thereby reducing the file size. This section discusses two implementations that are readily available in Spark ML to perform these tasks: PCA and SVD.

Principal Component Analysis (PCA)

Principal component analysis (PCA) can be used to identify patterns in large datasets. It can recognize the variables in your dataset that contain the most information. It is a linear representation of your dataset. Today, we have deep learning algorithms like autoencoder that do the same things and are useful for modeling complex, non-linear relationships. For the scope of this book, we focus on PCA. A general rule of thumb: Variables with distributions centered closely on the mean typically contain minimal information. The next section explains the PCA mechanics with a simple example.

Mechanics

Say you have six customers who bought the same car with the same model and make in different months of the year. You want to determine the top variable that can explain their satisfaction (see Table 4-1).

Table 4-1. *Sample Data*

Participant_ID	Quality	Reliability
1	10	6
2	9	4
3	8	5
4	3	3
5	2	2
6	1	1

When you use a single variable, `Quality`, and plot the data, you will observe that participants one to three will be close to each other on the upper end of the line, and the remaining three participants will be on the lower end. Even with a single data line, you can infer that the first three participants and the last three are similar. Plotting this data in a two-dimensional space shows that the first three participants cluster to the top right and the last three participants cluster to the bottom left (see Figure 4-10a). PCA will guide you in identifying the variable that is most important to the clustering of these variables. Let's embark on the journey of how PCA is calculated.

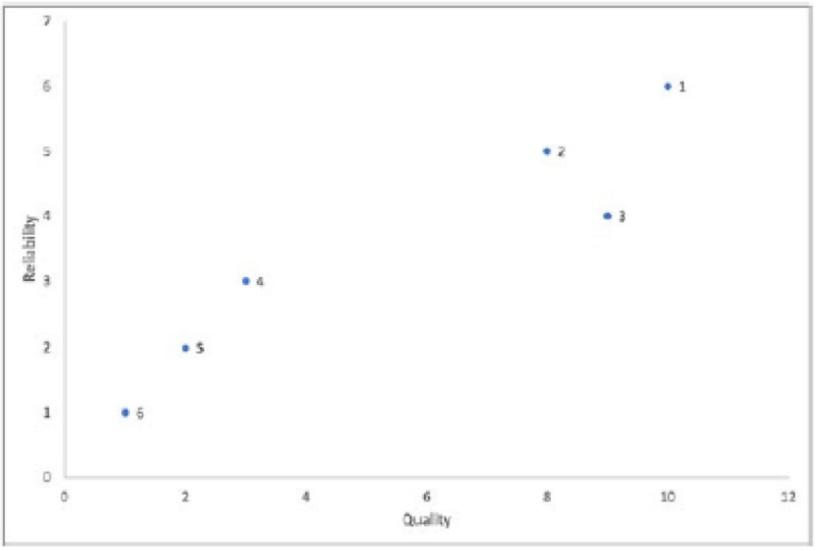


Figure 4-10a. *Car satisfaction data*

First, the means of both variables are calculated. The mean values of `Quality` and `Reliability` are 5.5 and 3.5, respectively. We shift the origin from (0,0) as shown in the graph in Figure 4-10b to 5.5 and 3.5. This is to make sure the data is centered on the origin.

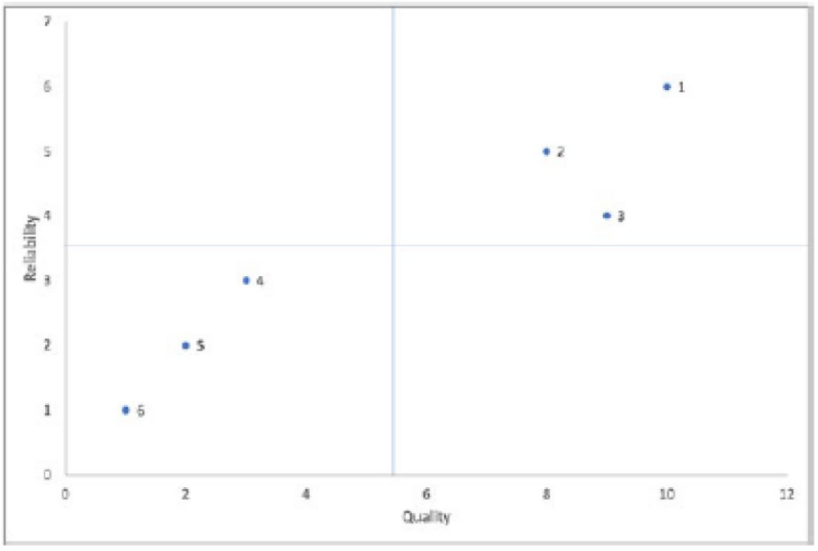


Figure 4-10b. *Car satisfaction data with axis*

Now we try to fit a line through these points until it fits the data well, passing through the origin. How can you define this criterion and know if a line fits closely? Well, you do know that when the projected distances from the data to the line are minimized, it is the closest fit (Point 5 is projected to the line passing through the origin). The same condition can be flipped and can be stated as a line that maximizes the distance from projected points to the origin (the dark shaded line in the same figure). PCA finds the best-fitting line by maximizing the sum of the squared distances from the projected points to the origin (see Figure 4-10c).

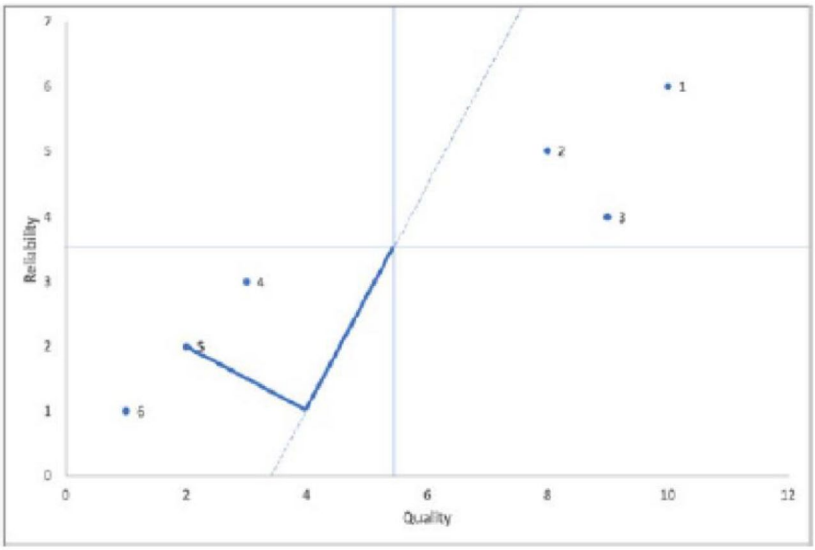


Figure 4-10c. *PCA projections*

The reason we use squared distances is to avoid any negative values. We start the process by rotating an arbitrary line through the origin and start calculating the distances from the origin to the projected points on the line. Figure 4-10c shows the distance of projected Point 5 to the origin. The highlighted dark line indicates the distance. Similar distances are calculated for the remaining points to arrive at the sum of the squared distances. All these distances are compared, and the line that has the *maximum sum of squared distance* is selected (see Figure 4-10d).

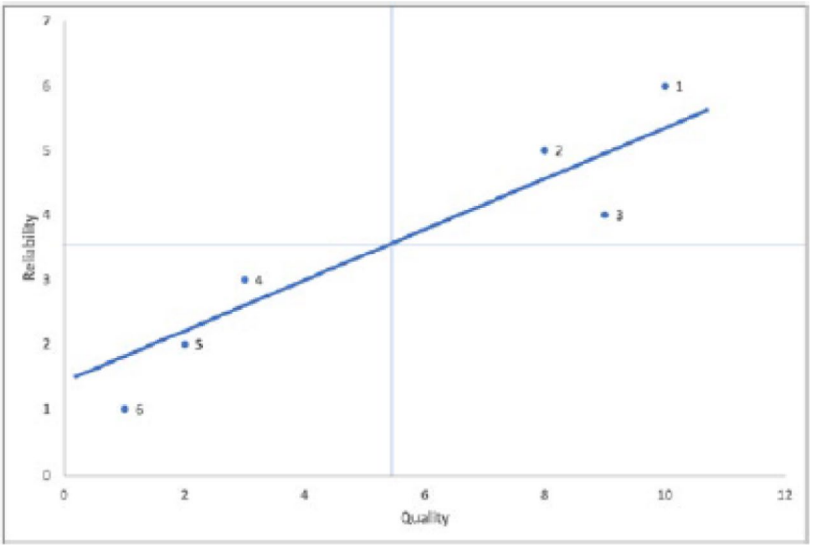


Figure 4-10d. *PCA line*

The indicated line maximizes the sum of the squared distance across all the points. So, this is considered PC1, or principal component 1. Assuming this line has a slope of 0.25, you can infer that for every four units you move in the quality direction, you will go up in reliability by one unit. This is a pseudo-indication that the data is all spread out on the quality axis. Therefore, quality plays a major role in defining PC1. This component can be thought of as a linear combination of four units of quality and one unit of reliability. PC2 is simply the line through the origin that is perpendicular to PC1. Its ratio of variables would be inverse to that of PC1. This component can be considered a linear combination of four reliability units and one quality unit. If you scale both of these to unit vectors, you can calculate the *loading scores for PCs*. *Eigenvalues* are the sum of squares of the distances between the projected points and the origins of the identified principal components. You can convert the *eigenvalues* of each principal component into variations by dividing them by `(sample size -1)`.

Here is the code for the PySpark implementation of PCA:

```
from pyspark.ml.feature import PCA
from pyspark.ml.linalg import Vectors
no_of_components = 3 #custom number
# Initialize PCA with specified parameters
```

```
pca = PCA(k=no_of_components, inputCol="features", outputCol="pcaFeatures")
model = pca.fit(df)
result = model.transform(df).select("pcaFeatures")
result.show(truncate=False)
```

pcaFeatures is the reduced set of features produced by the PCA algorithm that can be used in the downstream machine learning process. Notice that we ran with 16 input variables and ended up with three components. Essentially, we have reduced the dimension size by a factor of 1/5 (3/16 features). To get the loading scores for each variable, you can use the following code:

```
model.pc.toArray()
array([[-3.41021399e-04,  2.79524640e-04,  2.58353293e-03],
       [-9.99998245e-01,  1.83654726e-03,  1.13892524e-04],
       [-1.22934480e-05,  9.79995613e-04,  7.79347982e-03],
       [-1.83671689e-03, -9.99996986e-01, -7.36955549e-04],
       [ 1.48468991e-05,  1.01391994e-03,  2.75121381e-03],
       [-1.13085547e-04,  7.49207153e-04, -9.99889046e-01],
       [-1.26153895e-05, -6.36100089e-06, -1.04654388e-02],
       [-1.78789640e-05, -4.11817349e-05,  5.51411389e-04],
       [ 6.41085932e-06, -5.23364803e-05, -1.45349520e-04],
       [-1.11185424e-05,  1.30366514e-05,  2.01500982e-04],
       [ 2.91665702e-06,  4.42643869e-06,  3.95562163e-05],
       [-1.12221341e-05,  1.26153926e-05,  6.17569266e-04],
       [ 1.01623400e-05,  1.50687571e-05,  8.23933054e-05],
       [-5.68377754e-07,  6.95393403e-05,  1.03951369e-03],
       [-7.60886236e-05, -1.16754927e-04, -3.24662847e-03],
       [-8.55162111e-06, -6.01853226e-05, -4.94522998e-03]])
```

Now, let's map these loading scores to individual variables. We are using the features_list variable, which we fed earlier to the assembler function.

Table 4-2. *Mapping Loading Scores to Individual Variables*

Variable	PC1	PC2	PC3
age	-0.0003	0.0003	0.0026
balance	-1.0000	0.0018	0.0001
day	0.0000	0.0010	0.0078
duration	-0.0018	-1.0000	-0.0007
campaign	0.0000	0.0010	0.0028
pdays	-0.0001	0.0007	-0.9999
previous	0.0000	0.0000	-0.0105
job	0.0000	0.0000	0.0006
marital	0.0000	-0.0001	-0.0001
education	0.0000	0.0000	0.0002
default	0.0000	0.0000	0.0000
housing	0.0000	0.0000	0.0006
loan	0.0000	0.0000	0.0001
contact	0.0000	0.0001	0.0010
month	-0.0001	-0.0001	-0.0032
poutcome	0.0000	-0.0001	-0.0049

You can see that the `balance` variable dominates the PC1 component, `duration` the second PC2 component, and `pdays` the third PC3 component. You should be cautious about these results because the input data is not standardized. The continuous features are dominating the output of PCA since they have high variance. Another output of the algorithm you should focus on is the variance explained by each component. In PySpark, you can use the following code to do that:

```
model.explainedVariance
#Output: DenseVector([0.9918, 0.0071, 0.0011])
```

You can use the following code to plot these values as a graph. This graph is called a *Scree* plot.

```
import matplotlib.pyplot as plt
import numpy as np
x = []
for i in range(0, len(model.explainedVariance)):
#extract explained variance values& cumulative sums
    x.append('PC' + str(i + 1))
y = np.array(model.explainedVariance)
z = np.cumsum(model.explainedVariance)
plt.xlabel('Principal Components')
plt.ylabel('Variance Explained')
plt.bar(x, y)
plt.plot(x, z)
```

As you can see in Figure 4-10e, the first principal component explains 99 percent of the variance in the data. This is because we did not scale the DataFrame before we used PCA. So, let's do one more run with `StandardScaler` before using PCA. You will notice a big change in the variance explained plot.

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StandardScaler
#assemble and scale individual columns to one column - 'features2'
def scaled_assemble_vectors(df, features_list, target_variable_name):
    stages = []
    #assemble vectors

assembler = VectorAssembler(inputCols=features_list, outputCol='assembled_
features')

scaler = StandardScaler(inputCol=assembler.getOutputCol(),
outputCol='features2')
    stages = [assembler, scaler]
    #select all the columns + target + newly created 'features' column
    selectedCols = [target_variable_name, 'features2'] + features_list
    #use pipeline to process sequentially
    pipeline = Pipeline(stages=stages)
```

```
    #assembler model
    scaleAssembleModel = pipeline.fit(df)
    #apply assembler model on data
    df = scaleAssembleModel.transform(df).select(selectedCols)
    return df
features_list = df.columns
features_list.remove(target_variable_name)
df = scaled_assemble_vectors(df, features_list, target_variable_name)
from pyspark.ml.feature import PCA
from pyspark.ml.linalg import Vectors
# perform the pca on scaled features
pca = PCA(k=3, inputCol="features2", outputCol="pcaFeatures")
model = pca.fit(df)
result = model.transform(df).select("pcaFeatures")
result.show(truncate=False)
```

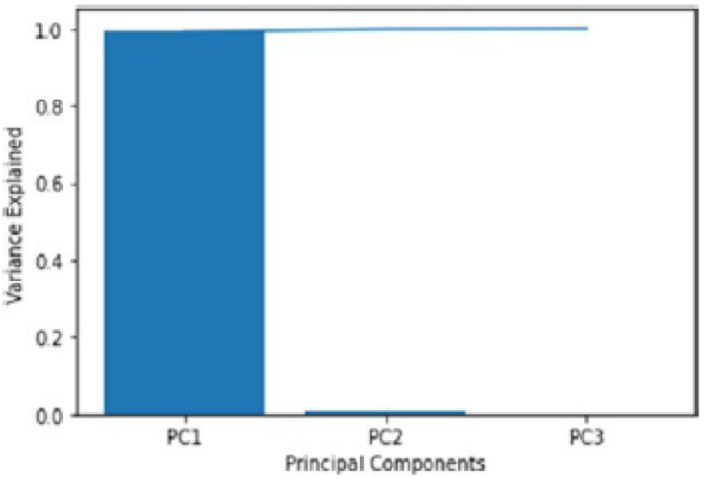


Figure 4-10e. *Variance explained*

When you run PCA with the newly created DataFrame from the preceding code, you see the variance explained by the first component changed drastically (see Figure 4-10f). You need to be aware of this scenario while using PCA, especially when dealing with continuous and categorical variables.

```
model.explainedVariance
#Output: DenseVector([0.1434, 0.0987, 0.0787])
```

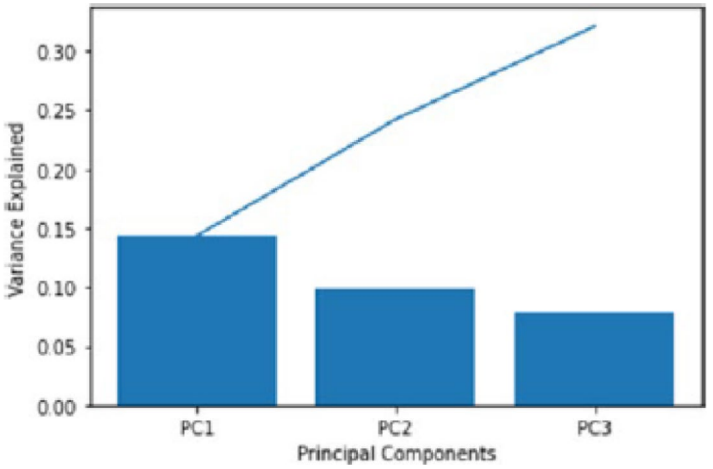


Figure 4-10f. *Variance explained*

Singular Value Decomposition (SVD)

The best-known method for matrix decomposition is Singular Value Decomposition, or SVD. Any matrix can be decomposed using SVD, thus making it a more stable algorithm. SVD factorizes a matrix into three matrices, U, Σ, and V, such that

$A = U\Sigma V^T$,

where

- A is the original matrix with r x c dimension. r – rows, c – columns.
- U is an r x r matrix, also called the left singular vectors of A.
- Σ is an r x c diagonal matrix. The diagonal values are the singular values of original matrix A.
- V^T is a c x c matrix, also called the right singular vectors of A.

Okay, how does this apply to the concept of variable selection? The answer lies in the Σ component. Let's say you have an image with 1,000 features. Using SVD, you can take the top k singular values by accessing the Σ matrix. You pick this k depending upon the amount of information you want to be represented in the lower dimension. In a PCA scenario, you can think of the variance explained. Assume the top 100 (k) out of the 1,000 features explain 80 percent of the information in the data, and you are okay

with that. Then, you will pick the top 100 singular values and substitute the rest of the values with 0 in the Σ matrix. The resulting low-rank matrix can then be used to create a representation of the original data, as shown here:

- U – r x k left singular values
- Σ – k x k singular values
- V^T – c x k right singular values

SVDs are used more for image data compression and in image recovery. Because of the matrix factorization, it also plays a huge role in decomposing a `user x item` matrix in a collaborative filtering recommendation system. Here's the PySpark code to implement SVD in this data:

```
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.linalg.distributed import RowMatrix
# convert DataFrame to rdd
# convert df to RDD and extract features as array
df_svd_vector = df.rdd.map(lambda x: x['features'].toArray())
#create a Rowmatrix from the RDD of vectors
mat = RowMatrix(df_svd_vector)
# Compute the top 5 singular values and corresponding singular vectors.
svd = mat.computeSVD(5, computeU=True)
U = svd.U       # The U factor is a RowMatrix.
s = svd.s       # The singular values are stored in a local dense vector.
V = svd.V       # The V factor is a local dense matrix.
```

You can read more about eigenvalue decomposition of a matrix to get a more in-depth understanding of SVD. Now, we move on to target-based variable reduction.

Built-in Variable Selection Process: With Target

This section explores supervised variable selection techniques.

ChiSq Selector

ChiSq stands for *chi-square*. It is also called the "goodness of fit" statistic, denoted by the symbol ***χ2.*** It uses the *chi-squared test of independence* to select the best features. This selection process is useful when dealing with categorical variables. If you prefer to use continuous variables, you need to bin them into groups before using this technique. We explain chi-square in detail using the example Titanic dataset (https://www.kaggle.com/c/titanic).

		Survived		
		No	Yes	Total
Gender	Female	81	231	312
	Male	468	109	577
	Total	549	340	889

Figure 4-11. *Titanic example: gender versus survived*

The table shown in Figure 4-11 is called a *contingency table*. By looking at the table, you can notice that the percentage of female survivors (231/340 = 67.9%) is much higher than the percentage of male survivors (109/340 = 32.0%). Does this mean there is a relationship between `Gender` and `Survived`? Well, you can answer this question using a chi-square test. In a chi-square test, you have null and alternate hypotheses as follows:

> *Null hypothesis: There is no relationship between the two variables.*
>
> *Alternate hypothesis: There is a relationship between the two variables.*

We use a 5 percent significance level for this test. The formula for chi-square is provided here:

$$\chi^2 = \sum \frac{(O-E)^2}{E}$$

χ2 – Chi-Square, O – Observed data, E – Expected data

Observed data refers to the data in the earlier table. Expected data is calculated based on the totals. The expected value for all four cells is provided in Table 4-3.

Table 4-3. *Expected Values*

Gender = Female and Survived = No	(312*549)/889 = 192.67
Gender = Female and Survived = Yes	(312*340)/889 = 119.32
Gender = Male and Survived = No	(577*549)/889 = 356.32
Gender = Male and Survived = Yes	(577*340)/889 = 220.67

We calculated chi-square using the formula shown earlier and summed the *chi-square* column to arrive at the total chi-square value in Table 4-4.

Table 4-4. *Chi-Square Value*

Cell	Observed	Expected	Chi-Square
Gender = Female and Survived = No	81	192.67	64.72
Gender = Female and Survived = Yes	231	119.32	104.51
Gender = Male and Survived = No	468	356.32	34.99
Gender = Male and Survived = Yes	109	220.67	56.51
		Total	260.75

The last step is to calculate the degrees of freedom, which is done using the following formula:

```
degrees_of_freedom or df = (r - 1) * (c - 1)
```

where

```
r = number of rows
c = number of columns
```

In this example, $r = 2$ (`Gender` has two rows) and $c=2$ (`Survived` has two columns). Therefore, the degrees of freedom is 1. Okay, we are all set. Now we can use the chisquare table (https://people.richland.edu/james/lecture/m170/tbl-chi.html) or an online chi-square calculator (https://stattrek.com/online-calculator/chi-square.aspx) to calculate the p-value. The p-value for this experiment is close to 0. It means that our results are significant, which means that we can reject the null hypothesis. Thus, we can conclude that there is a relationship between `Gender` and `Survived`.

Note Chi-square is susceptible to small frequencies. When the expected value in a cell is less than 5, it can lead to errors in conclusion.

How is this relevant to variable selection? In this example, we calculated a chi-square value of 260.75, which tells the relationship between two variables (`Gender` and `Survived`). When the relationship is high, the chi-square value will be high and vice versa. In a variable selection setting, you can pick variables with high chi-square values because they have a high relationship with the target and thus improve your prediction results.

Note Chi-square does not measure the strength of relationship. You should use correlation to determine the strength of the relationship.

The code to perform chi-square–based selection is provided next. You might have to rerun assembled vectors because we are only dealing with categorical variables for this type of selection.

```
#select only categorical features present in the dataset
features_list = char_vars #this option is used only for ChiSqselector
#run assembled vectors before running this code with the updated
features list
from pyspark.ml.feature import ChiSqSelector
from pyspark.ml.linalg import Vectors
#chisqselector instance initiation
selector = ChiSqSelector(numTopFeatures=6, featuresCol="features",
                         outputCol="selectedFeatures", labelCol="y")
#fit on the data
chi_selector = selector.fit(df)
#result datasets
result = chi_selector.transform(df)
#outputs
print("ChiSqSelector output with top %d features selected" % selector.
getNumTopFeatures())
```

```
print("Selected Indices: ", chi_selector.selectedFeatures)
features_df['chisq_importance'] = features_df['idx'].apply(lambda x: 1 if x
in chi_selector.selectedFeatures else 0)
print(features_df)
```

We have successfully implemented a chi-square–based feature-selection technique for categorical variables.

Model-based Feature Selection

Model-based feature selection requires you to build a machine learning model to find the important features. We talk about these models in detail in the next chapter. For now, imagine you want a computer to distinguish between *dogs* and *cats*. It is easy for a human to tell the difference, but it is tough for a computer to learn this on its own. That's where a machine learning model comes into the picture. The model teaches the computer to look for specific information to distinguish between dogs and cats. This specific information from which the model learns is called *features*. The more information a particular feature provides, the more important that feature is for the modeling process.

In PySpark, tree-based methods (Decision Tree, Random Forest, and Gradient Boosting) provide this option by default. This section focuses on the feature-importance aspect rather than the actual model. The following code calculates the feature importance. We used Random Forest in the bank dataset example. You can replicate the same with other algorithms.

```
from pyspark.ml.classification import RandomForestClassifier
rf = RandomForestClassifier(featuresCol='features', labelCol=target_
variable_name)
rf_model = rf.fit(df)
rf_model.featureImportances
#Output:
SparseVector(16, {0: 0.0381, 1: 0.0088, 2: 0.0048, 3: 0.4557, 4: 0.002, 5:
0.0347, 6: 0.0313, 7: 0.0021, 8: 0.0019, 9: 0.001, 10: 0.0001, 11: 0.0112,
12: 0.0017, 13: 0.0215, 14: 0.1484, 15: 0.2366})
```

That's it. We now have the feature importance scores. We can use the following code to format the output. Remember the `features_df` DataFrame we created in the vector assembler section? We will use it here.

```
import pandas as pd
features_df = pd.DataFrame()
for k, v in df.schema["features"].metadata["ml_attr"]["attrs"].items():
    final = pd.DataFrame(v)
    features_df=features_df.append(final,ignore_index=True)

#temporary output rf_output
rf_output = rf_model.featureImportances
features_df['Importance'] = features_df['idx'].apply(lambda x: rf_output[x]
if x in rf_output.indices else 0)
#sort values based on descending importance feature
features_df.sort_values("Importance", ascending=False, inplace=True)
```

We now have a nice-looking table, as shown in Figure 4-12. The most important feature is `duration`, followed by `poutcome`, and so on.

	idx	name	Importance
3	3	duration	0.439098
15	15	poutcome	0.265681
14	14	month	0.139936
5	5	pdays	0.046669
13	13	contact	0.030492
11	11	housing	0.027726
6	6	previous	0.012508
0	0	age	0.012080
1	1	balance	0.006088
2	2	day	0.005995
7	7	job	0.004189
8	8	marital	0.003499
12	12	loan	0.002518
4	4	campaign	0.002192
9	9	education	0.001166
10	10	default	0.000161

Figure 4-12. *Random Forest: feature importance*

In addition, you can use this table to create a feature importance plot, as shown in Figure 4-13.

```
import matplotlib.pyplot as plt
#just for plotting purposes sort it to ascending
features_df.sort_values("Importance", ascending=True, inplace=True)
```

```
plt.barh(features_df['name'], features_df['Importance'])
plt.title("Feature Importance Plot")
plt.xlabel("Importance Score")
plt.ylabel("Variable Importance")
```

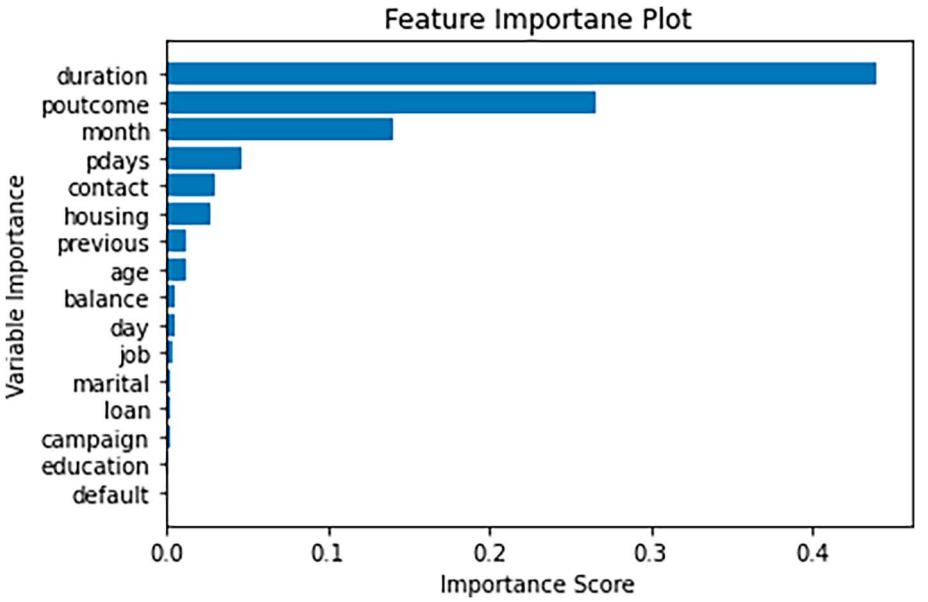


Figure 4-13. *Feature importance plot*

Pretty cool, right? You can use other algorithms to accomplish the same task. However, when dealing with large datasets, Random Forest–based feature selection makes the process a lot easier. One drawback to using tree-based methods is that it uses the training dataset to calculate the feature importance. This can inflate the importance of numeric and high-cardinality variables. To overcome this, you can use *permutation importance,* which uses the test dataset to calculate the feature importance. Permutation importance is out of scope for this book.

Note When you use linear models like logistic regression, use the coefficients from the model to showcase feature importance. The higher the *absolute value* of the coefficients, the more important the variable and vice versa.

Lastly, we focus on *continuous* and *multinomial* targets.

1) For continuous targets, you can use regression trees (`RandomForestRegressor` or other trees), which can be accessed from the `pyspark.ml.regression` module. You have to use the `LinearRegression` module instead of `LogisticRegression` for a continuous model. The rest of the discussed code can be reused to calculate feature importance.

2) For multinomial targets, you can still use the tree-based models discussed in this section, with the exception of the linear model `LogisticRegression`. It is a bit tricky to calculate feature importance using logistic regression when dealing with multinomial targets. We recommend you stick to tree-based models for such targets.

Until now, we have gone through the code without looking into the formula behind the calculation of feature importance. No worries. We revisit this section when discussing trees in the next chapter.

EXERCISE 4-2: MODEL-BASED SELECTION

You can use either the bank dataset or the housing dataset.

Question 1: Implement decision tree feature importance. Compare and contrast it with Random Forest output.

Question 2: Implement gradient-boosted tree feature importance. Compare and contrast it with Random Forest output.

Question 3: Implement logistic regression feature importance. Compare and contrast it with Random Forest output. (Hint: Use the coefficient of logistic regression. Note: Logistic regression produces dense vector instead of sparse vector.)

Custom-built Variable Selection Process

This section introduces some custom variable selection techniques.

Information Value Using Weight of Evidence

Weight of evidence (WOE) and information value (IV) are simple yet powerful techniques to perform *variable transformation and selection*. These concepts are closely connected to the logistic regression modeling technique. They are widely used in credit scoring to measure the separation of good versus bad customers. For example, if you are trying to predict whether customers will default on a payment, all the customers who have defaulted will be considered events and those who haven't can be considered nonevents.

The formula to calculate WOE and IV is provided here:

$$WOE = \ln\left(\frac{\text{Event\%}}{\text{Non Event\%}}\right)$$

$$IV = \sum\left(Event\% - Non\,Event\%\right) * \ln\left(\frac{\text{Event\%}}{\text{Non Event\%}}\right)$$

Or simply,

$$IV = \sum\left(Event\% - Non\,Event\%\right) * \left(WOE\right)$$

Figure 4-14 shows how to calculate these values.

Variable Name	Min. Value	Max. Value	Count	# Event	# Non Event	Event%	Non event%	WOE	Event% - Non event%	IV
Age	10	20	1200	150	1050	28.3%	19.0%	0.3992	9.3%	0.03718
Age	21	30	900	120	780	22.6%	14.1%	0.4733	8.5%	0.04040
Age	31	40	1090	110	980	20.8%	17.7%	0.1580	3.0%	0.00479
Age	41	50	1460	100	1360	18.9%	24.6%	-0.2650	-5.7%	0.01517
Age	50	inf	1410	50	1360	9.4%	24.6%	-0.9582	-15.2%	0.14525
Total			**6060**	**530**	**5530**					**0.24279**

Figure 4-14. *WOE and IV calculation*

The advantages of WOE transformation are as follows:

1. Handles missing values.
2. Handles outliers.
3. The transformation is based on the logarithmic value of distributions. This is aligned with the logistic regression output function.
4. There is no need for dummy variables.
5. Using the proper binning technique can establish a monotonic relationship (either increase or decrease) between the independent and dependent variables.

In addition, the IV value can be used to select variables based on the predictive power table shown in Figure 4-15.

Information Value (IV)	Predictive Power
< 0.02	useless for prediction
0.02 to 0.1	weak predictor
0.1 to 0.3	medium predictor
0.3 to 0.5	strong predictor
> 0.5	suspicious or too good to be true

Figure 4-15. *IV and predictive power*

As shown in Figure 4-14 for WOE and IV calculation, you need categories to perform the calculation. Now, you may be wondering, what should I do with continuous variables? As mentioned, a proper binning technique should be used to convert continuous to categorical before performing WOE and IV calculations. When the number of continuous variables is small (say less than ten), you can create your own custom bins and use them for WOE calculations. Even then, creating custom bins is a tedious task. Now, think about a large dataset with hundreds of continuous variables. It would be a nightmare to create custom bins. That's where monotonic binning comes into play.

Monotonic Binning Using Spearman Correlation

First, let's consider what *monotonic* means. See Figure 4-16.

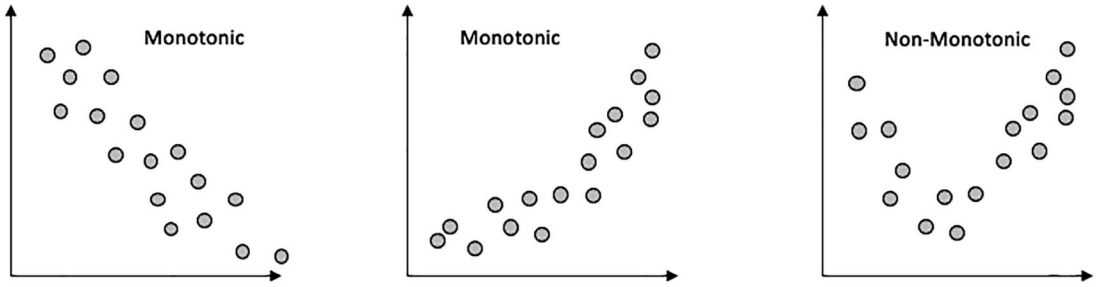


Figure 4-16. *Monotonic versus non-monotonic*

As you can see in Figure 4-16, in the first two scenarios, as the x-axis variable increases, the y-axis value has a single trend: increasing or decreasing. This is a monotonic relationship. In the third scenario, the x-axis variable change does not have a single trend: increasing or decreasing behavior. Therefore, it is non-monotonic. A monotonic relationship is beneficial when it comes to explanatory purposes. You can easily explain the results to a nontechnical audience when two variables exhibit a monotonic relationship. Let's say, for example, the `balance` and `target` variables have a relationship, as shown in Figure 4-17.

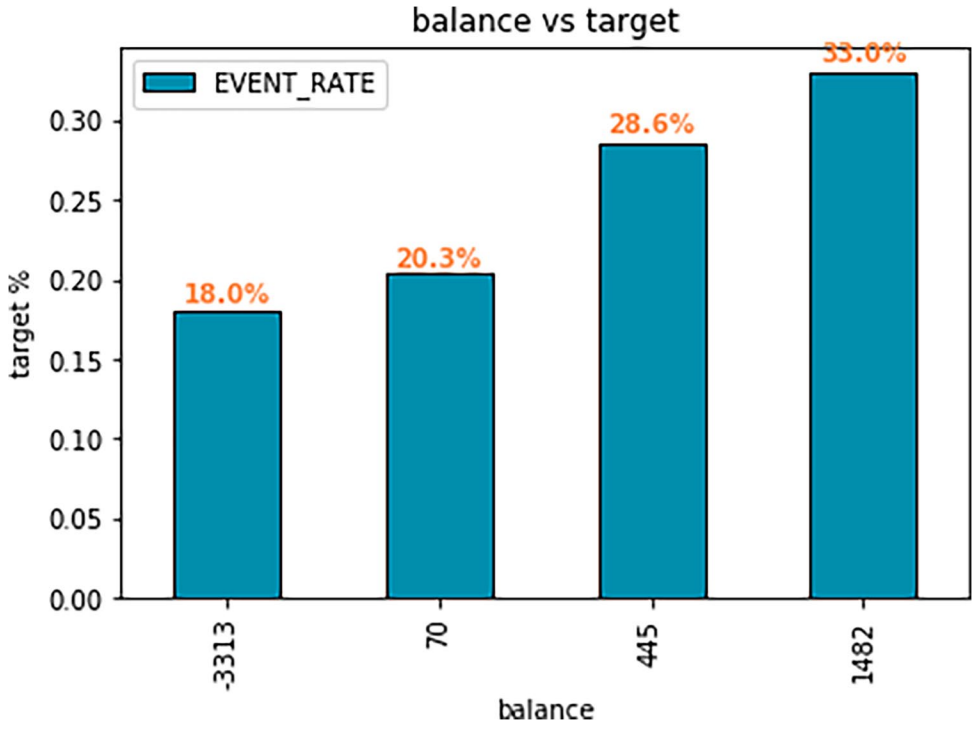


Figure 4-17. *Balance versus target bins*

Looking at this graph, you can easily say, "As the balance increases, the target conversion rate increases." This result is easy for a non-technical audience to interpret.

To perform monotonic binning, you need to understand the Spearman correlation. The Spearman correlation measures the strength and direction of the monotonic association between two variables. Okay, how does it differ from the Pearson correlation? You test for a linear relationship between two variables in Pearson correlation, but Spearman measures monotonic relationships. In this case, the second graph in Figure 4-16 would have a moderate linear relationship for Pearson correlation. However, the Spearman correlation would consider it a strong monotonic relationship. The Spearman correlation is also known as the *rank-order correlation*.

How Do You Calculate the Spearman Correlation by Hand?

The formula for the Spearman correlation is provided here:

$$\rho = 1 - \frac{6\sum d_i^2}{n\left(n^2 - 1\right)}$$

Here, ρ is the Spearman correlation, d_i is the difference in paired ranks, and n is the number of cases. When you have tied ranks, you need to use the following formula. Rank is the ordered number in a result set.

$$\rho = \frac{\sum_i \left(x_i - \bar{x}\right)\left(y_i - \bar{y}\right)}{\sqrt{\sum_i \left(x_i - \bar{x}\right)^2 \sum_i \left(y_i - \bar{y}\right)^2}}$$

Here, i is the paired score. For now, we use only the first formula to calculate the Spearman correlation (see Figure 4-18).

Mathematics (mark)	Physics (mark)	Rank for Mathematics	Rank for Physics	d_i	d_i^2
48	56	7	7	0	0
75	89	3	2	1	1
53	65	6	6	0	0
68	72	5	5	0	0
92	74	1	4	-3	9
88	81	2	3	-1	1
74	99	4	1	3	9
				Total	20

Figure 4-18. *Spearman correlation calculation*

Using d_i^2 as 20 and n as 7, you can calculate the ρ value using the first equation:

```
p = 1 - ((6*20)/(7*((7*7) - 1))) = 0.64285714
```

What do we infer from this number 0.64? In general, a Spearman correlation value ranges from -1 to +1. When the value is close to 1 or -1, the monotonic relationship between the two variables is stronger. When the value is close to 0, there is no monotonic relationship between the two variables. In this example, mathematics and physics grades exhibit a strong monotonic relationship. Figure 4-19 shows a guide for determining the strength of the relationship.

Absolute ρ value	Relationship
0.00 to 0.19	very weak
0.20 to 0.39	weak
0.40 to 0.59	moderate
0.60 to 0.79	strong
0.80 to 1.0	very strong

Figure 4-19. *Spearman correlation value guide*

How Is Spearman Correlation Used to Create Monotonic Bins for Continuous Variables?

In the code for monotonic binning, notice that the `mono_bin()` function uses a `max_bin` option. This is a user-specified value. For continuous variables, we start with the `max_bin` provided by the user and decrease the bins as each step proceeds. We create quantiles

based on the max_bin. Next, we calculate the Spearman correlation based on the newly created quantile bins and the target variable. Until we establish a strong monotonic relationship between the continuous variable and the target variable, we keep on decreasing the max_bin value to create new quantiles and calculate the Spearman value. When we reach a strong correlation between the two variables, the binning process is stopped.

Here is the complete code to calculate WOE and IV, including the monotonic binning option for continuous variables. We have provided the entire code from data load to WOE and IV calculation. This code can be used on any dataset. And, the code is provided as a self-study.

```
# Default parameters
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
dataset = 2 # 1 or 2
# Load Dataset
from pyspark.sql import functions as F
# if the option is set as 1 load bank dataset
if dataset == 1:
    target_variable_name = 'y'
    # File location and type
    file_location = '/FileStore/tables/bank_full.csv'
    file_type = 'csv'

    # CSV options
    infer_schema = 'true'
    first_row_is_header = 'true'
    delimiter = ';'

    # The applied options are for CSV files. For other file types, these
    will be ignored.
    df = spark.read.format(file_type) \
    .option('inferSchema', infer_schema) \
    .option('header', first_row_is_header) \
    .option('sep', delimiter) \
    .load(file_location)
    df = df.withColumn(target_variable_name, F.when(df[target_variable_
```

```
name] == 'no', 0).otherwise(1))
# if the option is not set as 1 load housing dataset
else:
    #filename = "melb_data.csv"
    target_variable_name = 'type'
    file_location = '/FileStore/tables/melb_data.csv'
    file_type = 'csv'

    # CSV options
    infer_schema = 'true'
    first_row_is_header = 'true'
    delimiter = ','

    # The applied options are for CSV files. For other file types, these
    will be ignored.
    df = spark.read.format(file_type) \
    .option('inferSchema', infer_schema) \
    .option('header', first_row_is_header) \
    .option('sep', delimiter) \
    .load(file_location)
    df = df.withColumn(target_variable_name, F.when(df[target_variable_
    name] == 'h', 0).otherwise(1))
df.show()
# target check
df.groupby(target_variable_name).count().show()
#identify variable types and perform some operations
def variable_type(df):
    vars_list = df.dtypes
    char_vars = []
    num_vars = []
    for i in vars_list:
        if i[1] in ('string'):
            char_vars.append(i[0])
        else:
            num_vars.append(i[0])
    return char_vars, num_vars
```

```
char_vars, num_vars = variable_type(df)
if dataset != 1:
    char_vars.remove('Address')
    char_vars.remove('SellerG')
    num_vars.remove('Date')
    char_vars.remove('Suburb')
num_vars.remove(target_variable_name)
final_vars = char_vars + num_vars
# WOE & IV code
from pyspark.sql import functions as F
import pandas as pd
import numpy as np
from pyspark.ml.feature import QuantileDiscretizer
from pyspark.ml.feature import VectorAssembler
import scipy.stats.stats as stats
#default parameters
#rho value for Spearman correlation. You can adjust this value
custom_rho = 1
#maximum number of bins to start off with. It will keep decreasing from
this number. Adjustable parameter
max_bin = 20
# This function calculates the WOE and IV values based on the
PySpark output.
# Note: This portion of the code is Python-based implementation for
efficiency purposes
def calculate_woe(count_df, event_df, min_value, max_value, feature):
    # implementing the table structure shown in WOE & IV calculation figure
    woe_df = pd.merge(left=count_df, right=event_df)
    woe_df['min_value'] = min_value
    woe_df['max_value'] = max_value
    woe_df['non_event'] = woe_df['count'] - woe_df['event']
    woe_df['event_rate'] = woe_df['event']/woe_df['count']
    woe_df['nonevent_rate'] = woe_df['non_event']/woe_df['count']
    woe_df['dist_event'] = woe_df['event']/woe_df['event'].sum()
    woe_df['dist_nonevent'] = woe_df['non_event']/woe_df['non_event'].sum()
```

```
    woe_df['woe'] = np.log(woe_df['dist_event']/woe_df['dist_nonevent'])
    woe_df['iv'] = (woe_df['dist_event'] -
    woe_df['dist_nonevent'])*woe_df['woe']
    woe_df['varname'] = [feature]* len(woe_df)
    woe_df = woe_df[['varname','min_value', 'max_value', 'count', 'event',
    'non_event', 'event_rate', 'nonevent_rate', 'dist_event','dist_
    nonevent','woe', 'iv']]
# replace infinite values with zero for iv calculation
    woe_df = woe_df.replace([np.inf, -np.inf], 0)
# sum of iv values for features
    woe_df['iv'] = woe_df['iv'].sum()
    return woe_df
#monotonic binning function implemented along with Spearman correlation
def mono_bin(temp_df, feature, target, n = max_bin):
    r = 0
    while np.abs(r) < custom_rho and n > 1:
        try:
            #
Quantile discretizer cuts data into equal number of observations
            qds =
QuantileDiscretizer(numBuckets=n, inputCol=feature, outputCol='buckets',
relativeError=0.01)
            bucketizer = qds.fit(temp_df)
            temp_df = bucketizer.transform(temp_df)
            #
create corr_df is Python-based implementation for efficiency purposes;
group by and calculate avg for feature and target
            corr_df =
temp_df.groupby('buckets').agg({feature:'avg', target:'avg'}).toPandas()
            corr_df.columns = ['buckets', feature, target]
            r, p =
stats.spearmanr(corr_df[feature], corr_df[target]) #spearman correlation
# decrement bin count if correlation condition is not met
            n = n - 1
        except Exception as e:
            n = n - 1
```

```
        return temp_df
#execute WOE for all the variables in the dataset
def execute_woe(df, target):
    count = -1
    for feature in final_vars:
        #
execute if the feature is not a target column name. Provided as an
extra check.
        if feature != target:
            count = count + 1
            temp_df = df.select([feature, target])
            #
perform monotonic binning for numeric variables before doing WOE
calculation
            if feature in num_vars:
                temp_df = mono_bin(temp_df, feature, target, n = max_bin)
                # group buckets in numerical
                grouped = temp_df.groupby('buckets')
            else:
                # just group categories in categorical
                grouped = temp_df.groupby(feature)
            #count and event value for each group

count_df = grouped.agg(F.count(target).alias('count')).toPandas()
            event_df = grouped.agg(F.sum(target).alias('event')).toPandas()
            #
store min and max values for variables. for category both takes the
same value.
            if feature in num_vars:

min_value = grouped.agg(F.min(feature).alias('min')).toPandas()['min']

max_value = grouped.agg(F.max(feature).alias('max')).toPandas()['max']
            else:
                min_value = count_df[feature]
                max_value = count_df[feature]
```

```
            #calculate WOE and IV
            temp_woe_df =
calculate_woe(count_df, event_df, min_value, max_value, feature)
            #final dataset creation
            if count == 0:
                final_woe_df = temp_woe_df
            else:
                final_woe_df =
final_woe_df.append(temp_woe_df, ignore_index=True)
        # separate IV dataset creation
        iv = pd.DataFrame({'IV':final_woe_df.groupby('varname').iv.max()})
        iv = iv.reset_index()
    return final_woe_df, iv
# invoke WOE & IV code
output, iv = execute_woe(df, target_variable_name)
```

That is all. You are now done with weight of evidence and information value.

Custom Transformers

So far, the chapter has been looking at some high-level APIs available in Spark ML that can be used to perform variable selection (built-in selection techniques discussed earlier). This section creates a custom Spark ML transformer to accomplish the same thing. Why do we need to discuss custom transformers?

Take a moment to think about the `RandomForest` code used for variable selection earlier:

```
from pyspark.ml.classification import RandomForestClassifier
rf = RandomForestClassifier(featuresCol='features', labelCol=target_
variable_name)
rf_model = rf.fit(df)
rf_model.featureImportances
```

With a few lines of code, we did the following: (1) invoked the `RandomForest`; (2) defined an instance of `RandomForest` object; (3) fit the model on the dataset; and (4) showed the importance score. This is the power of object-oriented programming (OOP). Here is a quick refresher on OOP concepts:

1. *Class* – A user-defined blueprint that is used to create an object instance. In line 2, `RandomForestClassifier` is a class and `rf` is the object instance.

2. *Methods* – A function written inside a class. In line 3, the `fit` function is a method.

3. *Abstraction* – This is a concept by which only the essential information is shown to the users. This is called implementation hiding. In the `RandomForestClassifier` code shown earlier, we only provide the input variable, target, and the dataset to fit the model. We don't see any details on how the Random Forest model is implemented in the background.

4. *Encapsulation* – This is a concept in which you hide information from users. This is called information hiding. Any *variable/method* can be hidden using this technique. The *variable/method* can be *protected* (single underscore in the front, cannot be accessed outside a class, accessible within the class and its subclass), or *private* (double underscore in the front, accessible only inside the class). When you look at the source code, you might still be able to see the variables/methods. So, they are conceptually hidden, but not visually hidden.

5. *Inheritance* – This is a concept in which you inherit the characteristics of a parent class to create a child class. Think of a square object. It is technically a rectangle with all four sides equal. In this case, `rectangle` is the parent class and `square` is the child class.

6. *Polymorphism* – This is a concept in which you can morph the characteristics of an object. It means that you can make the same object do multiple things other than the usual. Think of the *addition (+)* symbol. Originally, it can sum only numeric variables. In Python, when you include addition between strings, it concatenates them. When you execute this code—`print("Hello " + "World")`—it concatenates the string to produce the `Hello World` output. In this case, the *addition (+)* symbol can add strings as well as numerics. This is an example of polymorphism.

Using OOP concepts, you can create a powerful machine learning class object. This makes it simple for end users to use your own code. These objects are also easier to combine into a single pipeline or workflow, thereby creating an efficient end-to-end process. The next section quickly examines the main concepts used to create such custom code.

Main Concepts in Pipelines

DataFrame: A collection of different datatypes organized in a single ML dataset.

Estimator: This is used to *fit* a function to a DataFrame. In `RandomForestClassifier`, the `fit` function is the estimator.

Transformer: This is used to *transform* a DataFrame from one to another. Sometimes, a transformer requires a function to be fit before it can be applied. In `RandomForestClassifier`, the `predict` function is a transformer.

Pipeline: An ML workflow or sequence of steps to get to the end result.

Parameter: The estimator and transformer use uniform API for specifying parameters.

This example creates a custom correlation `class` and uses it for variable selection. We will demonstrate the pipeline use case. Before you see any implementation, let's quickly go through the code used to calculate correlation on a dataset.

```
from pyspark.mllib.stat import Statistics
import pandas as pd
correlation_type = 'pearson' # 'pearson', 'spearman'
#transformer function
features_df = pd.DataFrame()
for k, v in df.schema["features"].metadata["ml_attr"]["attrs"].items():
    final = pd.DataFrame(v)
    features_df=features_df.append(final,ignore_index=True)
column_names = list(features_df['name'])
df_vector = df.rdd.map(lambda x: x['features'].toArray())
```

```
#generate correlation-matrix using  Statistics.corr
matrix = Statistics.corr(df_vector, method=correlation_type)
corr_df = pd.DataFrame(matrix, columns=column_names, index=column_names)
```

The preceding code produces a pandas DataFrame with the correlation results. We will add some more code to extend its functionality. We include an option to find the most correlated variables using an external parameter.

```
#transformer continuation
final_corr_df = pd.DataFrame(corr_df.abs().unstack().sort_
values(kind='quicksort')).reset_index()
final_corr_df.rename({'level_0': 'col1', 'level_1': 'col2', 0:
'correlation_value'}, axis=1, inplace=True)
final_corr_df = final_corr_df[final_corr_df['col1'] != final_corr_
df['col2']]
correlation_cutoff = 0.65 #custom parameter
final_corr_df[final_corr_df['correlation_value'] > correlation_cutoff]
```

We recommend you get the feel of the preceding code before you use the custom `class`.

```
# Import the estimator and transformer classes
from pyspark.ml import Transformer
# Parameter sharing class. We will use this for input column
from pyspark.ml.param.shared import HasInputCol
# Statistics class to calculate correlation
from pyspark.mllib.stat import Statistics
import pandas as pd
# custom class definition
class CustomCorrelation(Transformer, HasInputCol):
    """
    A custom function to calculate the correlation between two variables.
    Parameters:
    -----------
    inputCol: default value (None)
```

```
Feature column name to be used for the correlation purpose. The input
column should be assembled vector.
    correlation_type: 'pearson' or 'spearman'
    correlation_cutoff: float, default value (0.7), accepted values 0 to 1

Columns more than the specified cutoff will be displayed in the output
dataframe.
    """
    # Initialize parameters for the function

def __init__(self, inputCol=None, correlation_type='pearson', correlation_
cutoff=0.7):
        super(CustomCorrelation, self).__init__()
        assert inputCol, "Please provide a assembled feature column name"
        #self.inputCol is class parameter
        self.inputCol = inputCol
        assert correlation_type == 'pearson' or correlation_type ==
        'spearman', "Please provide a valid option for correlation type.
        'pearson' or 'spearman'. "
        #self.correlation_type is class parameter
        self.correlation_type = correlation_type

assert 0.0 <= correlation_cutoff <= 1.0, "Provide a valid value for cutoff.
Accepted range is 0 to 1"
        #self.correlation_cutoff is class parameter
        self.correlation_cutoff = correlation_cutoff
    #
Transformer function, method inside a class, '_transform' - protected
parameter
    def _transform(self, df):
features_df = pd.DataFrame()
        for k, v in df.schema[self.inputCol].metadata["ml_attr"]["attrs"].
        items():
            final = pd.DataFrame(v)
            features_df=features_df.append(final,ignore_index=True)
 column_names = list(features_df['name'])
 df_vector = df.rdd.map(lambda x: x[self.inputCol].toArray())
```

```
  #self.correlation_type is class parameter
  matrix = Statistics.corr(df_vector, method=self.correlation_type)
  # apply pandas dataframe operation on the fit output

corr_df = pd.DataFrame(matrix, columns=column_names, index=column_names)

final_corr_df = pd.DataFrame(corr_df.abs().unstack().sort_
values(kind='quicksort')).reset_index()

final_corr_df.rename({'level_0': 'col1', 'level_1': 'col2', 0:
'correlation_value'}, axis=1, inplace=True)

final_corr_df = final_corr_df[final_corr_df['col1'] != final_corr_
df['col2']]
        #shortlisted DataFrame based on custom cutoff

shortlisted_corr_df = final_corr_df[final_corr_df['correlation_value'] >
self.correlation_cutoff]
# Return the full correlation matrix and shortlisted ones
        return corr_df, shortlisted_corr_df
```

Voila! You now have a custom correlation `class`, which works similarly to any Spark ML object. This code can be saved in a separate Python file called `customcorrelation.py`. Let's test this new class.

```
from customcorrelation import CustomCorrelation
clf = CustomCorrelation(inputCol='features')
output, shorlisted_output = clf.transform(df)
```

That's all. You used three lines of code to calculate correlation among the input features in your dataset. This is similar to the `RandomForestClassifier` invocation you saw before. Some might question whether we can do the same with *user-defined functions (udfs)*. You are right in terms of abstraction of code; however, *udfs* fail to fit in the pipeline workflow. All this effort so far makes the work pipeline compatible. So, let's test that feature too. To demonstrate this feature, we combine the assemble vector step and correlation step into one pipeline. If you need to rerun the code, you need to execute until "Step 2: Apply StringIndexer to Character Columns" discussed earlier in this chapter.

```
from pyspark.ml.feature import VectorAssembler
from customcorrelation import CustomCorrelation
from pyspark.ml import Pipeline
# remove feature vectors
df = df.drop(*['features2','features'])
#exclude target variable and select all other feature vectors
features_list = df.columns
features_list.remove(target_variable_name)
stages = []
#assemble vectors
assembler = VectorAssembler(inputCols=features_list, outputCol='features')
custom_corr = CustomCorrelation(inputCol=assembler.getOutputCol())
stages = [assembler, custom_corr]
#use pipeline to process sequentially
pipeline = Pipeline(stages=stages)
#pipeline model
pipelineModel = pipeline.fit(df)
#apply pipeline model on data
output, shorlisted_output = pipelineModel.transform(df)
```

Pretty cool! Now, you can implement your own custom transformers and create a neat ML workflow.

Voting-based Selection

So far, we have discussed multiple techniques for the variable selection process. Wouldn't it be great to combine all the variable selection options into one process? You can accomplish this using a voting-based schema. The idea is to apply multiple techniques to select variables. When a variable selection technique picks a variable, you assign a score to the variable. In other words, the technique voted for that variable. At the end, you calculate the total score for each variable and select the variables with the highest scores (see Figure 4-20). This way, you end up picking the variables with the highest "preference" from a variety of selection techniques, which, depending on the case at hand, might not work well when used individually.

Variable Name	Random Forest	Decision Tree	Chi Square	Gradient Boosting	WOE & IV	Total Votes
Age	1	1	1	0	1	4
Gender	1	1	0	1	1	4
Income	1	0	0	1	1	3
Education	0	0	1	1	0	2
Occupation	1	0	0	0	0	1
Homeowner	0	0	0	0	0	0

Figure 4-20. *Voting-based selection*

This section requires you to complete Exercise 4-2: Model-Based Selection. For the sake of demonstration, we provide the solution for the exercise here (see Figure 4-21). We use all the target-based selection techniques discussed earlier in this chapter.

```
#decision tree algorithm – solution to Question 1
from pyspark.ml.classification import DecisionTreeClassifier
dt = DecisionTreeClassifier(featuresCol='features', labelCol=target_
variable_name)
dt_model = dt.fit(df)
dt_output = dt_model.featureImportances
features_df['Decision_Tree'] = features_df['idx'].apply(lambda x: dt_
output[x] if x in dt_output.indices else 0)

#Gradient boosting algorithm – solution to Question 2
from pyspark.ml.classification import GBTClassifier
gb = GBTClassifier(featuresCol='features', labelCol=target_variable_name)
gb_model = gb.fit(df)
gb_output = gb_model.featureImportances
features_df['Gradient Boosting'] = features_df['idx'].apply(lambda x: gb_
output[x] if x in gb_output.indices else 0)
#logistic regression algorithm – solution to Question 3
from pyspark.ml.classification import LogisticRegression
```

```
lr = LogisticRegression(featuresCol='features', labelCol=target_
variable_name)
lr_model = lr.fit(df)
lr_output = lr_model.coefficients
#absolute value is used to convert the negative coefficients. This should
be done only for feature importance.
features_df['Logistic Regression'] = features_df['idx'].apply(lambda x:
abs(lr_output[x]))
#random forest - technique which we discussed before
from pyspark.ml.classification import RandomForestClassifier
rf = RandomForestClassifier(featuresCol='features', labelCol=target_
variable_name)
rf_model = rf.fit(df)
rf_model.featureImportances
rf_output = rf_model.featureImportances
features_df['Random Forest'] = features_df['idx'].apply(lambda x: rf_
output[x] if x in rf_output.indices else 0)
#voting-based selection
features_df.drop('idx', axis=1, inplace=True)
num_top_features = 7 #top n features from each algorithm
columns = ['Decision_Tree', 'Gradient Boosting', 'Logistic Regression',
'Random Forest']
score_table = pd.DataFrame({},[])
score_table['name'] = features_df['name']
#evaluate top features by each algo
for i in columns:
    score_table[i] = features_df['name'].isin(list(features_
df.nlargest(num_top_features,i)['name'])).astype(int)
# sum up the final score
score_table['final_score'] = score_table.sum(axis=1)
score_table.sort_values('final_score',ascending=0)
```

	name	Decision_Tree	Gradient Boosting	Logistic Regression	Random Forest	final_score
15	poutcome	1	1	1	1	4
14	month	1	1	1	1	4
3	duration	1	1	0	1	3
11	housing	0	1	1	1	3
5	pdays	1	1	0	1	3
2	day	1	1	0	1	3
13	contact	1	1	1	0	3
0	age	1	0	0	1	2
12	loan	0	0	1	0	1
8	marital	0	0	1	0	1
10	default	0	0	1	0	1
6	previous	0	0	0	0	0
1	balance	0	0	0	0	0
4	campaign	0	0	0	0	0
7	job	0	0	0	0	0
9	education	0	0	0	0	0

Figure 4-21. *Voting-based selection output*

Summary

- You went through variable selection techniques in detail.
- You learned about target- versus non-target–based selection techniques.
- You created your own Spark ML transformer and fit it with the ML workflow.
- Finally, you combined all the selection techniques to create a vote-based selection technique.

Great job! In the next chapter, you learn about the most exciting part of this book—machine learning models. Keep learning, and stay tuned.

CHAPTER 5

Supervised Learning Algorithms

It's time to do some learning based on the data. Most people think machine learning is applying an algorithm to given data and predicting results. Well, it's not just that simple. Eighty percent of the work involves data collection, preprocessing, cleaning, feature engineering, transformation, and selecting the best features. The remaining 20 percent is spent on building machine learning models, validation, and deployment. The entire operation is called *MLOps* (*machine learning operations*). It is similar to DevOps, but for machine learning. In order to understand and deploy a production model, you should be familiar with each component in MLOps. The previous chapters discussed the 80 percent of the work. If you skipped those chapters, we recommend reading them before you read this chapter. In addition, business and domain knowledge helps you improve the process throughout.

Machine Learning Algorithm Basics

This section discusses machine learning algorithms. These algorithms are broadly categorized into three types: supervised, unsupervised, and reinforcement learning (see Table 5-1).

© Ramcharan Kakarla, Sundar Krishnan, Balaji Dhamodharan and Venkata Gunnu 2024
R. Kakarla et al., *Applied Data Science Using PySpark*, https://doi.org/10.1007/979-8-8688-0820-3_5

Table 5-1. *Types of Machine Learning Algorithms*

	Supervised	**Unsupervised**	**Reinforcement**
What is it?	Algorithm is trained on labeled data.	Algorithm is trained on unlabeled data.	Agent is trained based on the rewards earned for each action.
Where is it used?	Regression, classification	Clustering, segmentation, association	Rewards-based
What type of data?	Labeled data. Example: Predict cat vs. dog or predict house price	Unlabeled data; just input features. Example: Customer profiling	No predefined data. Example: Self-driving cars, game agent
Is supervision required?	Yes	No	No
What is the goal?	Reduce the error in target prediction using the known input/output mapping	Understand the pattern and behaviors in the data to tell a story	Trial and error method. Make the agent learn based on its errors and to improve rewards in the long run.

Apart from the categories mentioned, two other types of algorithms are common these days: recommendation systems and semi-supervised algorithms. A recommendation system can be either supervised or unsupervised. You learn more about this in the next chapter. A semi-supervised algorithm combines the supervised and unsupervised learning approaches. This approach uses a small amount of labeled data to classify the large amount of unlabeled data. Semi-supervised algorithms and reinforcement learning are out of the scope of this book. This chapter covers supervised learning algorithms. In the next chapter, we cover unsupervised and recommendation algorithms.

Within supervised algorithms, you have these types: regression and classification (binary, multinomial, and multi-label).

Regression

In a regression model, you predict continuous values, such as house price. Regression is a statistical method used in finance, investing, and other disciplines that attempts to determine the strength and character of the relationship between one dependent variable (usually denoted by Y) and a series of other variables (known as independent variables).

Classification

In a classification model, we predict categorical values, such as cats versus dogs, spam versus not spam, and so on. When the number of categories is two, it is called binary classification. When the number of categories is more than two, it is called multinomial. Binary classification models are a subset of multinomial classification models. Another subset of multinomial classification models is ordinal classification. In an ordinal model, the targets are ordered. For example, predicting the likelihood of purchasing an economy ticket versus a business ticket versus a first-class ticket. An important thing to remember is that the target is mutually exclusive in these classification models. In some cases, there can be multiple targets for a single observation, such as the curation of articles. This book can be curated into these titles: "machine learning, data science, PySpark applications." Any of these three is a valid target. This type of task is a multi-label classification.

In either case (regression or classification), the goal is to reduce the error between the predicted and actual values. We start with the goal (reduce error) to cover some related concepts and then dive into algorithms.

Loss Functions

Loss functions are the error functions associated with each supervised learning algorithm (see Table 5-2). The goal is to try to minimize the loss function in any model-training process. Why minimize it? Consider an example of predicting the weight of a person. The original weight values are 30, 40, and 50. Let's assume one of the models predicts the weights as 32, 46, and 57, respectively.

Table 5-2. *Types of Loss Functions*

	Name	Equation	When to use?
Regression	Squared error loss	(y – y-hat)^2	Commonly used loss function in regression. This loss function is easily differentiable.
	Absolute error loss	|y - y-hat|	Use absolute loss when outliers are present in data as it weights them equally.
	Pseudo-Huber loss	x = $\begin{cases} (y - y - hat)^2 ; when\ x \le \alpha \\ y - y - hat ; \lvert otherwise \end{cases}$	Combines both squared error and absolute error loss. Tune hyperparameter α to get best results.
Classification	Binary cross-entropy loss	-(y log(p) + (1-y) log(1-p))	Binary targets. Default loss function.
	Categorical cross-entropy loss	C - number of classes $-\sum_{x \in C} p(x) \log q(x)$	Multinomial targets. Default loss function.
	KL divergence loss	C - number of classes $-\sum_{x \in C} p(x) \log \left(\frac{q(x)}{p(x)} \right)$	Multinomial targets *Note: Cross-entropy = Entropy + KL Divergence*
	Hinge loss	x = y* y-hat max (0, 1 – x)	Binary targets. Typically used in support vector machines. Used to maximize margin between classes.

Using the squared error loss function, the total error for this model is as follows:

$$\text{Model 1 error} = (30 - 32)^2 + (40 - 46)^2 + (50 - 57)^2 = 4 + 36 + 49 = 89$$

A second model predicts the weights as 31, 42, and 52. For this model, the squared error is as follows:

$$\text{Model 2 error} = (30 - 31)^2 + (40 - 42)^2 + (50 - 52)^2 = 1 + 4 + 4 = 9$$

As you can see, when you start to minimize the squared error function, the prediction starts to come close to the original values. Therefore, understanding the loss function and choosing the appropriate one is an important task in a model-training process. You can visualize the loss function here: https://academo.org/demos/3d-surface-plotter/. We showcase an example using the squared error loss in Figure 5-1.

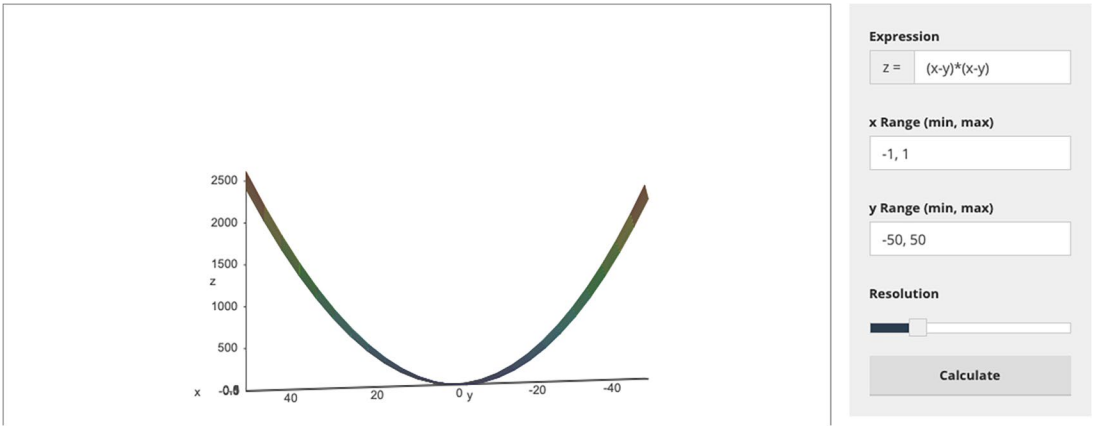


Figure 5-1. *Squared error loss*

When you reach the bottom of the loss function, the squared error value is minimal. In a model-training process, the model starts with an error and takes subsequent steps to reach the bottom value of the curve. This process is called *gradient descent.* Another loss function—absolute error loss—is shown in Figure 5-2.

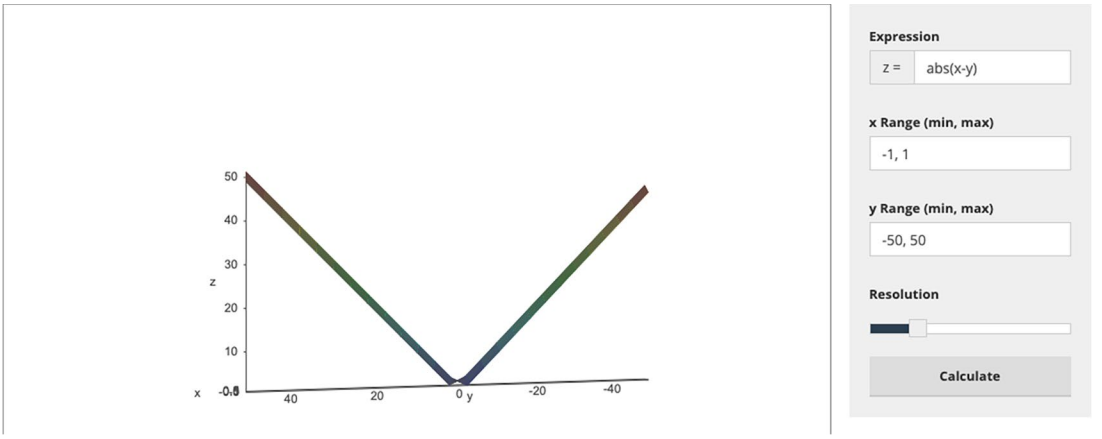


Figure 5-2. *Absolute error loss*

As we explore the algorithms in later sections, we switch the loss functions depending on the model.

Optimizers

Optimizers are used to find the minimum value of a function. In the previous section, we discussed that the model error is minimized when the loss function reaches the minimum value. Since we are trying to find the minimum value of the loss function, we can use optimizers to solve these tasks. The model parameters that produce the minimal error are called *optimal parameters*. The whole path is depicted in Figure 5-3.

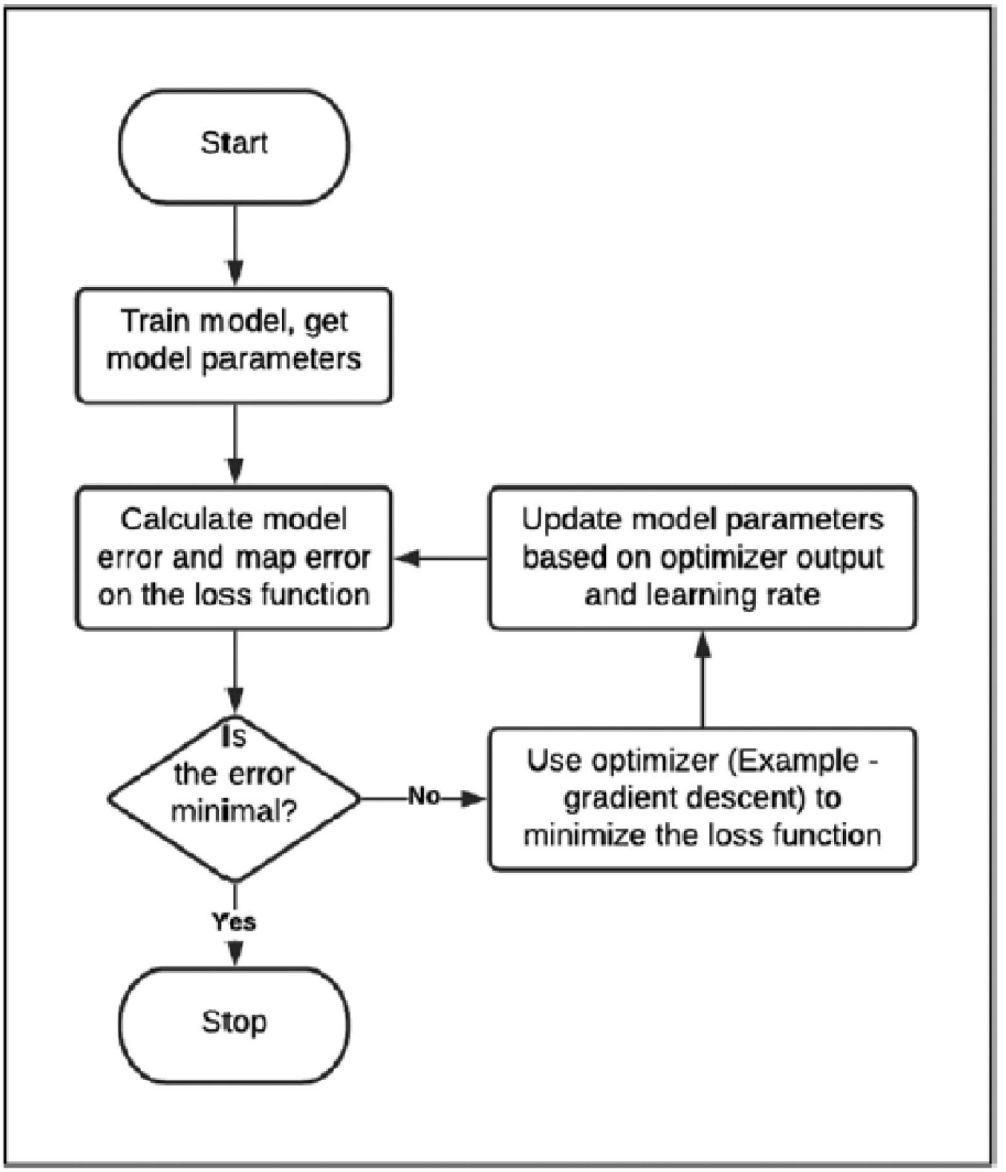

Start
Train model, get model parameters
Calculate model error and map error on the loss function
Update model parameters based on optimizer output and learning rate
Is the error minimal?
No
Use optimizer (Example - gradient descent) to minimize the loss function
Yes
Stop

Figure 5-3. *Optimizer flow chart*

In this section, we dive deep into different optimizers to accomplish this task. Let's start with gradient descent.

Gradient Descent

Imagine you are at the top of the mountain and your goal is to reach the bottom of the mountain using the shortest way possible. You won't jump directly from the top, because you don't know the height of the mountain. It would be crazy and you would most probably die. The unknown knowledge—*"Where is the bottom of the mountain?"*—is the function you are trying to solve here. To find this solution, you start descending the mountain by taking *small steps* in the *right direction*. These gradual movements allow you to adapt to the terrain while ensuring continuous descent. You repeat the process (small steps and continuous descent) several times to get to the bottom of the mountain. This whole idea encapsulates the concept of gradient descent (see Figure 5-4).

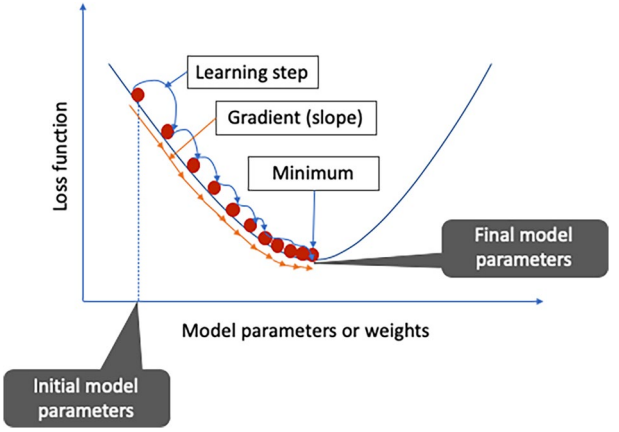


Figure 5-4. *Gradient descent*

Mathematically, the gradient is the slope value of the function the optimizer tries to solve. Hence, it is called gradient descent. It is shown using the orange line in Figure 5-4. The model parameters are updated using this equation:

$$\beta_i = \beta_i - \alpha * \text{Slope}_i$$

Where

$$\beta_i = \text{Model parameter of a variable } i$$

$$\alpha = \text{Learning rate}$$

$$\text{Slope}_i = \text{Gradient or slope with respect to the variable } i = \frac{\partial}{\partial \beta_i} \text{ (Error)}$$

In the initial model training phase, the slope is large and starts decreasing as you reach the optimal solution. You can stop the model-training process when one of the following conditions is satisfied:

1. Slope (gradient) reaches close enough to zero. At this point, the model parameter updates yield the same result.
2. The number of iterations or epochs is met.
3. The model improvement falls below a specified threshold. To accomplish this, you compare the previous iteration against the current iteration's performance to determine the model (delta) improvement and check whether the delta value is below the specified threshold.

Choosing Learning Rate

You saw the learning rate being used to update model parameters. The question is, "What is the best learning rate?"

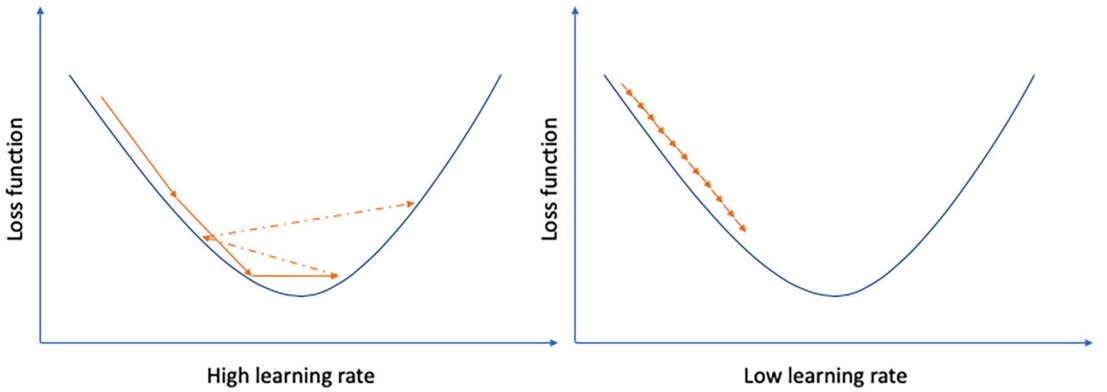


Figure 5-5. *Learning rate*

- By choosing a high learning rate, you descend much faster and could potentially miss the bottom of the curve. This is shown using the dotted lines in Figure 5-5. In this case, the model starts to produce chaotic results.

- However, by choosing a low learning rate, your model will take more time to converge on the optimal solution.

When your model is not producing the expected results, decrease the learning rate and rerun it. This is a hyperparameter that can be tuned according to your data and model needs. Another approach is to use an adaptive learning rate. In this case, the learning rate changes according to the slope of the curve. During the initial model training, it starts with a high learning rate (when slope is high), and as it descends the slope, the learning rate can be decreased. This way, you can reach the optimal solution much faster and you can get the best of both worlds (high and low learning rates). The adaption function can be linear or an order of magnitude. We discuss adaptive learning rate in more detail in a later section.

Local Minima Versus Global Minima

In Figure 5-6, you can see that the actual bottom of the curve is the global minimum position. Using gradient descent, you would not always be able to reach the bottom of the curve. Depending upon the initial parameters, the model could start descending from the left or the right. If it started from the right, it would have a high chance of reaching the global minimum. However, if it started from the left, it could get stuck in the local minimum, and the gradient descent operation would stop. This is a limitation of gradient descent. It cannot differentiate between local minimum and global minimum. You can avoid this problem by restarting the model-training process with different hyperparameters.

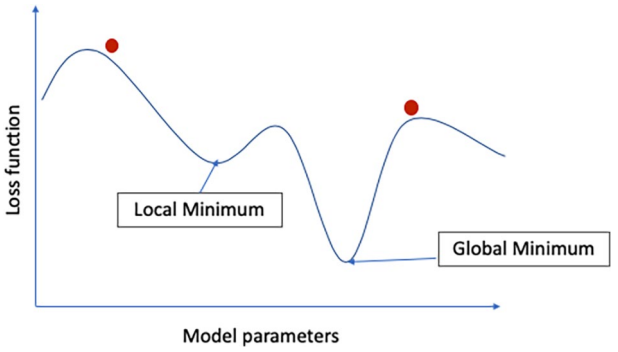


Figure 5-6. *Local minima versus global minima*

Model-training Time

As you know by now, the parameter update happens after all the samples in your training data are run through the gradient descent optimization process. This approach is okay for smaller datasets. However, when dealing with millions of records, gradient descent optimization is very slow. The model's training time increases drastically as the amount of data increases.

Stochastic/Mini-batch Gradient Descent

To solve the model-training time and the local minimum problem, we look at a variant of gradient descent. In stochastic gradient descent (SGD), the parameter update is done for every observation in the dataset. If your dataset has 100 rows, the parameter update happens 100 times in a single pass. This improves the training time and model convergence when the dataset size is huge. In addition, the randomness introduced by the SGD process helps the model escape the local minimum and reach the global (or better) minimum. Even though SGD has these advantages, there are some drawbacks too:

- Due to the randomness introduced each time, the optimizer takes noisy steps to converge to the minimum. Sometimes, these noisy steps could throw you off-course, and you could end up in one of the local minimums.
- Frequent SGD updates are computationally expensive.
- SGD cannot perform vectorized updates, since it uses one observation at a time.

Another variant of gradient descent is mini-batch gradient descent. Let's say you have 100 rows in the training data. In a single pass, gradient descent uses all 100 rows to perform one weight update, whereas SGD does a weight update for each observation (so 100 times). Is there a middle ground? Mini-batch gradient descent is the solution to this question. In a mini-batch GD, based on the batch size (let's say, 5), the weight update happens after each batch is processed. In this example, it happens 20 (100 rows /5 batch size = 20) times. An optimal batch size could help the model train and converge faster on the global minimum. Depending on the size of the data, a good default batch size is 32. Clearly, mini-batch GD overcomes the disadvantages listed in SGD. Still, mini-batch GD could sometimes converge on the local minimum instead of the global minimum.

Momentum

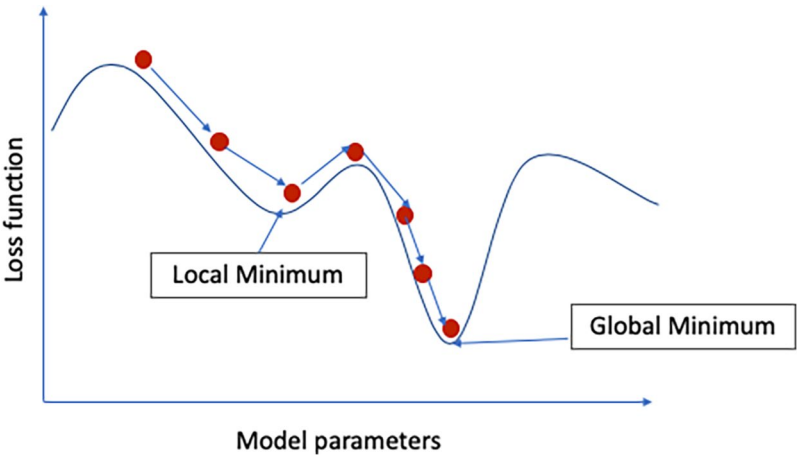


Figure 5-7. *Momentum*

Let's look at the equation for a momentum optimizer and then talk about the details (see Figure 5-7).

$$\beta_i = \beta_i - \alpha * \text{Slope}_i(t) + m = GD + m$$

$$m = w * (-\alpha * \text{Slope}_i(t-1))$$

where

i = Model parameter of a variable i

(t) = Current update

GD = Gradient descent

m = Momentum value

w = Weightage provided the previous update (typically 0.9)

(*t* – 1) = Previous update

To overcome the local minimum problem, you can extend the gradient descent equation by adding a momentum component. You can think of momentum as rolling a ball downhill. The ball gains momentum as it rolls down the hill. Even when the ball hits a small dip along the way (local minimum), it will still keep on rolling until it reaches a flat surface (global minimum). Without momentum, the ball would have rolled with a constant speed and would have ended up in the small dip (local minimum). With momentum, you incorporate the knowledge that the ball is moving downhill. This is done by adding the speed at which the ball was rolling in the previous step. The weightage component (w) determines the amount of weightage to be given for the previous step. The w is a hyperparameter that can be tuned for best results.

AdaGrad (Adaptive Gradient) Optimizer

You can also escape the local minimum by using an adaptive learning rate. We discussed this concept earlier; you see the nitty-gritty details in this section. The adaptive gradient is achieved by dividing the learning rate by an exponentially increasing parameter (G_i). This is shown in the following equation:

$$\beta_i = \beta_i - \frac{\propto}{\sqrt{G_i(t)} + \varepsilon} * \text{Slope}_i(t)$$

And G_i is given by

$$G_i(t) = G_i(t-1) + \left(\text{Slope}_i(t)\right)^2$$

At initial model training, $G_i(0) = 0$.

The slope term is always positive, and therefore $G_i(t)$ keeps on increasing in each iteration. Since you are dividing the learning rate by an increasing number, the learning rate keeps on decreasing. This is the whole concept behind AdaGrad. The epsilon (ε) parameter is set to a very small value to avoid division by zero. This ensures that even if Gi(t) becomes extremely small, the division remains stable, allowing AdaGrad to function reliably.

Root Mean Square Propagation (RMSprop) Optimizer

RMSprop is a variant of the AdaGrad optimizer. Look at the following equation:

$$\beta_i = \beta_i - \frac{\propto}{\sqrt{G_i(t)} + \varepsilon} * \text{Slope}_i(t)$$

The update rule is still the same. However, the change is introduced in the G_i parameter, as follows:

$$G_i(t) = \gamma G_i(t-1) + (1-\gamma)\left(\text{Slope}_i(t)\right)^2$$

At initial model training, $G_i(0) = 0$.

The difference between RMSprop and AdaGrad comes from the Gamma parameter (γ). This is another hyperparameter that can be optimized. Typically, the value for γ is set to 0.9. The γ parameter introduces a moving average to the gradient component. Since you take the square root of the moving average, this optimizer is called the *Root Mean Square optimizer*.

Adaptive Moment (Adam) Optimizer

Adaptive Moment combines RMSprop and Momentum, hence the name. This can help achieve momentum and a learning rate schedule (adaptive learning rate) by using one optimizer. The updated equation is provided here:

$$\beta_i = \beta_i - \frac{\propto}{\sqrt{G_i(t)} + \varepsilon} * m_i(t)$$

where $m_i(t)$ is the momentum component and is calculated using the equation

$$m_i(t) = w * m_i(t-1) + (1-w) * \text{Slope}_i(t)$$

At initial model training, $m_i(0) = 0$.

w is the weightage parameter used in the Momentum equation. Thus, $m_i(t)$ is the first moment estimate and $G_i(t)$ is the second moment estimate.

Quick Recap

- Error functions are also called *loss functions*.
- When you reach the bottom of a loss function, the model error is at its minimum. At this point, the model fits the data better.
- You can minimize the error function using optimizers such as gradient descent.
- Local minimum and global minimum issues can be solved either by adding momentum or tweaking the learning rate. In some cases, you can change the momentum and the learning rate simultaneously.

Activation Functions

Another component in an algorithm is the activation function. A simple way to think about an activation function is to look at biological neurons. When the body needs some energy, the brain activates the food neuron. The person then thinks, "I want to eat food." The brain fires these activations to communicate with the person and make them take action based on it. See Figure 5-8.

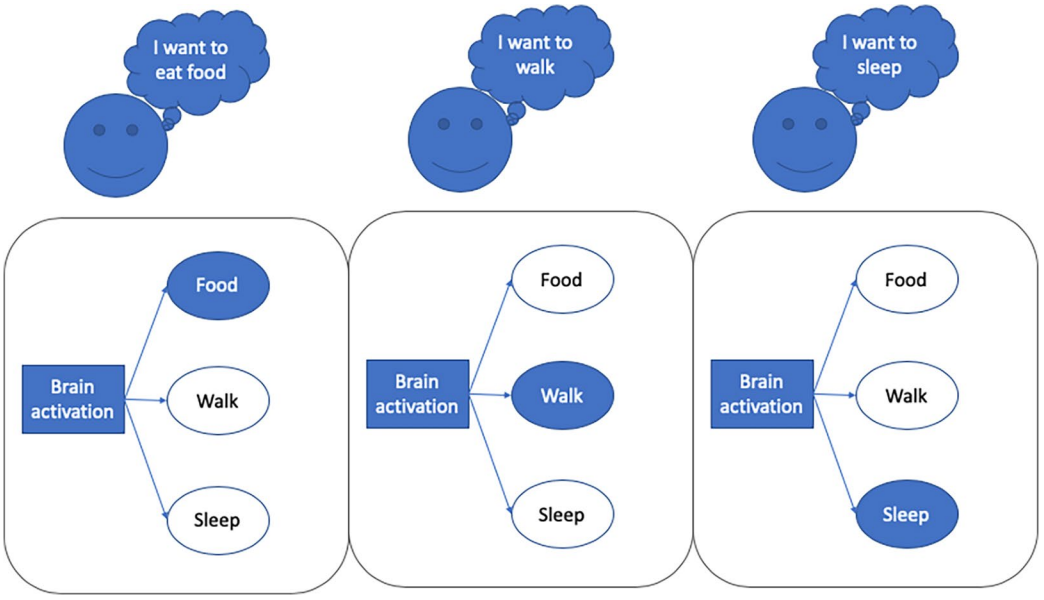


Figure 5-8. *Activation function illustration*

In a similar fashion, activation functions can be used to turn on/off certain actions that an algorithm performs when it is being trained. The following sections examine each one in detail.

Linear Activation Function

In a linear activation, the input data is passed without any change. It is represented by the equation

$$f(x) = x$$

where x is the input given to the activation function. The derivative of this function, $f(x)$, equals 1. The derivative values are useful while performing gradient descent updates in backpropagation. We discuss backpropagation in a later section. A linear function cannot learn a non-linear data type (see Figure 5-9). The next sections discuss non-linear functions, starting with Sigmoid.

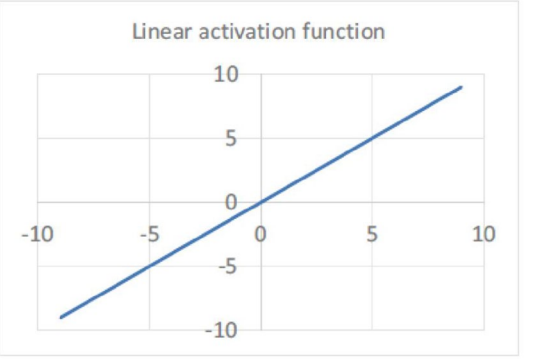


Figure 5-9. *Linear activation function*

Sigmoid Activation Function

In a Sigmoid function, the output is squished between 0 and 1 using the following equation (see Figure 5-10):

$$f(x) = \frac{1}{1 + \exp(-x)}$$

You can convert the $f(x)$ value to an on/off signal using a customized cut-off, as follows:

$$\begin{cases} 1 \ for \ f(x) \geq cut-off \\ 0 \ for \ f(x) < cut-off \end{cases}$$

When it comes to binary classification, most algorithms use 0.5 as the cut-off value. You can tweak it depending on your specific requirements. The derivate of this function is

$$f'(x) = f(x)(1-f(x))$$

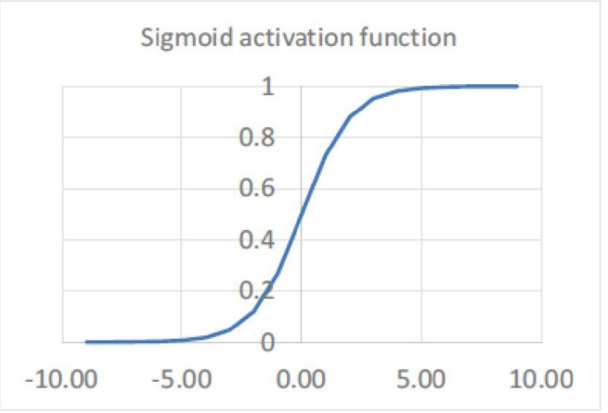


Figure 5-10. *Sigmoid function*

Hyperbolic Tangent (TanH) Function

In a TanH function, the output is squished between -1 and 1 using the following equation (see Figure 5-11):

$$f(x) = tanh(x) = \left(\frac{2}{1+\exp(-2x)}\right) - 1$$

Similar to the Sigmoid function, you can use a custom cut-off (default 0) to convert the output to a 0/1 binary output. The derivative of this function is

$$f'(x) = 1 - f(x)^2$$

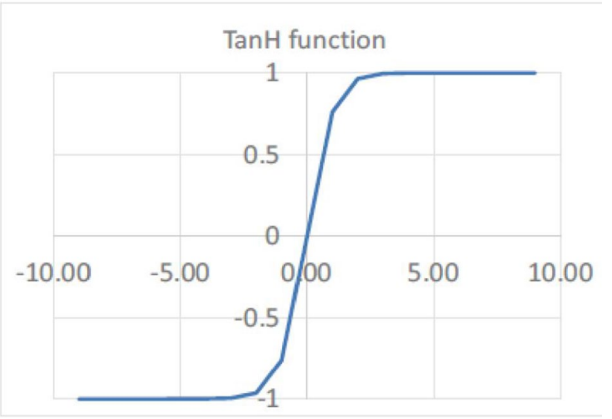


Figure 5-11. *TanH function*

Rectified Linear Unit (ReLu) Function

Rectified Linear Unit (ReLu) will output the input directly if it is positive (see Figure 5-12). Otherwise, it will output zero. The rectified linear activation function overcomes the vanishing gradient problem, allowing models to learn faster and perform better.

Figure 5-12. *ReLu function*

One of the problems with the Sigmoid and TanH functions is that the derivative tends to become zero for some deep models. The model parameter update will not happen because of this issue. ReLu can be used to solve this issue. In a ReLu function, for negative values of x the output is zero and for positive values of x the output is the same as x.

$$f(x)=\begin{cases} x \ \ for \ \ x \geq 0 \\ 0 \ \ for \ \ x < 0 \end{cases}$$

The derivate of this function is

$$f'(x)=\begin{cases} 1 \ \ for \ \ x \geq 0 \\ 0 \ \ for \ \ x < 0 \end{cases}$$

Leaky ReLu or Parametric ReLu Function

Leaky ReLu is similar to the ReLu function, but with a small tweak. The reason for the tweak is because, sometimes, the ReLu function may encounter a "dying ReLu" problem, where some neurons can become permanently inactive during training.

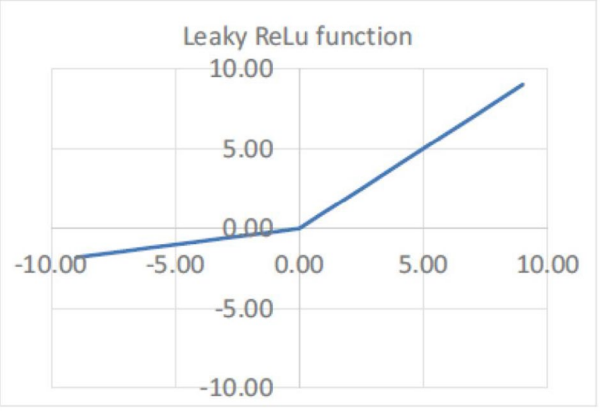


Figure 5-13. *Leaky ReLu*

In a Leaky ReLu function, for negative values of *x*, the output is a constant factor of *x* and for positive values of *x*, the output is same as *x* (see Figure 5-13), where α is the constant factor.

$$f(x)=\begin{cases} x \ \ for \ \ x \geq 0 \\ \alpha x \ \ for \ \ x < 0 \end{cases}$$

The derivative of this function is

$$f'(x)=\begin{cases} 1 \ \ for \ \ x \geq 0 \\ \alpha \ \ for \ \ x < 0 \end{cases}$$

Swish Activation Function

This is a relatively new activation function. It was introduced in 2017 by the Google team. It is shown in research that, by replacing the ReLu function with the Swish function, the classification accuracy is improved in the `ImageNet` model by 0.9 percent and in the `Inception_ResNet-v2` model by 0.6 percent (see Figure 5-14). The equation for the function is provided here:

$$f(x) = x * sigmoid(x) = x * \left(\frac{1}{1 + exp(-x)} \right)$$

The derivative of this function is given by

$$f'(x) = f(x) + sigmoid(x) * (1 - f(x))$$

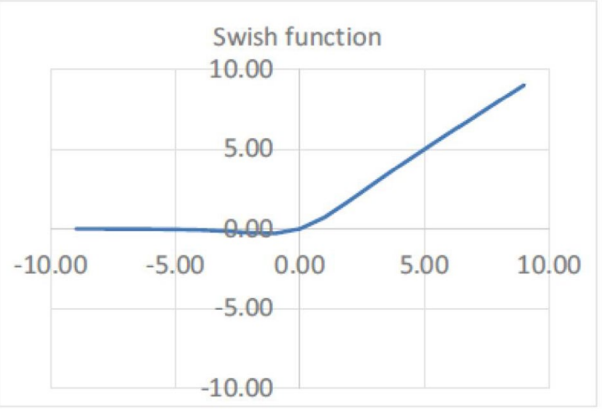


Figure 5-14. *Swish function*

Softmax Function

The equation for this function is

$$f(x) = \frac{e^{Xi}}{\sum_{j} e^{Xj}}$$

When the number of classes is three and the x values are [5, 4, 3], $f(x)$ is calculated as

$$f(5) = \frac{e^5}{e^5 + e^4 + e^3} = 0.665$$

For other classes, the values are 0.24 and 0.09, respectively. This activation function is used for multi-label classification problems. The class with the higher probability is chosen as the final output. In this case, the model will pick "Class 1" as output.

Batch Normalization

We've discussed the advantages of using mini-batch in a model-training process. However, it becomes tricky when the distribution of the inputs varies across batches. This will cause the algorithm to train a little bit longer to arrive at the optimal solution. You can resolve this problem using batch normalization. The concept is simple. The input data is standardized for each batch, and the model is trained on the standardized batch data. This will make the model's learning process more stable, and the algorithm will converge much faster. This concept is more powerful while training deep neural networks.

Dropout

Sometimes a model can overemphasize a single input and leave the rest out in the training process. This could lead to overfitting (we explore this concept in the next chapter) and introduce bias in the learning process. It happens a lot when you are dealing with continuous and binary input variables to train the model. Let's say you have `age` and `gender` as input. In this case, the model would give more weight to age since it has more unique values and would not give similar weight to gender. To overcome this problem, you can use a regularization technique called *dropout*. During training, some of the hidden layer (neural network) outputs are randomly dropped. This way, the algorithm will not always rely on the `age` variable to predict the output, thereby avoiding bias and overfitting.

Supervised Machine Learning Algorithms

You have now learned all the relevant concepts. This section dives deep into each of the machine learning algorithms. This entire section uses the bank dataset. The following starter code is used for all the machine learning algorithms.

```
# read the data
filename = "bank-full.csv"
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
data = spark.read.csv(filename, header=True, inferSchema=True, sep=';')
data.show()
# assemble feature vectors
from pyspark.ml.feature import VectorAssembler
# assemble individual columns to one column - 'features'
def assemble_vectors(df, features_list, target_variable_name):
    stages = []
    #assemble vectors
    assembler = VectorAssembler(inputCols=features_list,
outputCol='features')
    stages = [assembler]
    #select all the columns + target + newly created 'features' column
    selectedCols = [target_variable_name, 'features'] + features_list
    #use pipeline to process sequentially
    pipeline = Pipeline(stages=stages)
    #assembler model
    assembleModel = pipeline.fit(df)
    #apply assembler model on data
    df = assembleModel.transform(df).select(selectedCols)
    return df
```

Linear Regression

Linear regression is used to predict continuous outcomes, such as house prices or a customer's lifetime value. This algorithm assumes that the relationship between inputs and outputs is linear. When you use one input to predict the output, it is called *simple*

linear regression. When you use multiple inputs to predict the output, it is called *multiple linear regression*. The concept is shown in Figure 5-15.

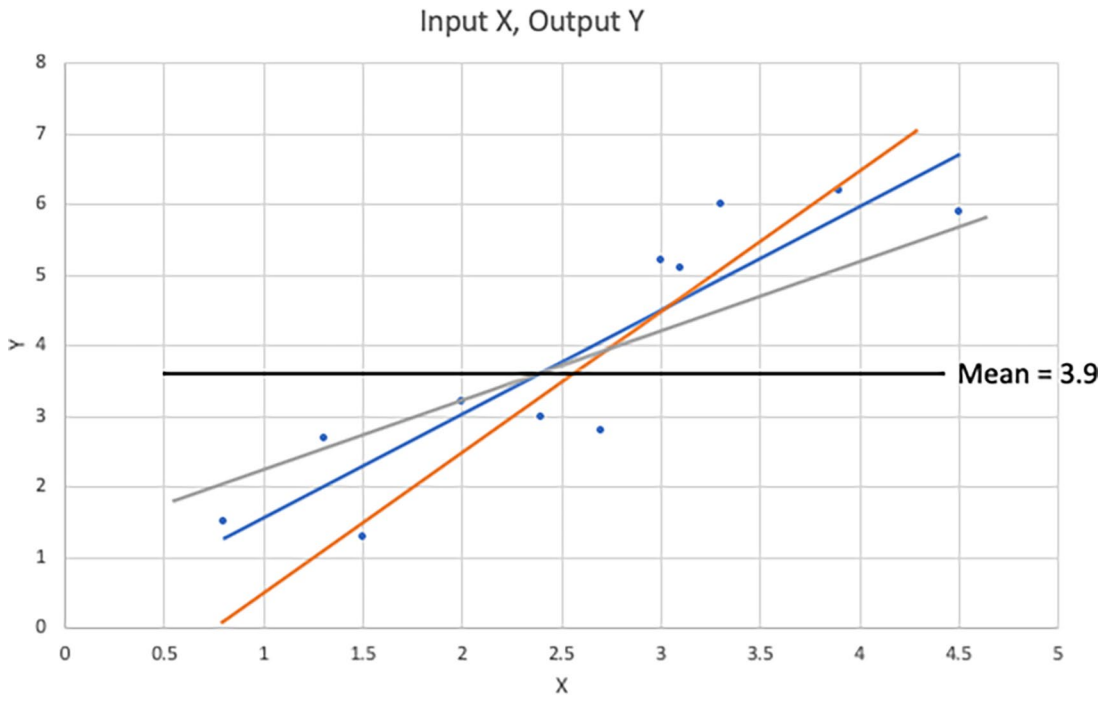


Figure 5-15. *Linear regression*

Looking at the figure, which line better fits the data? The goal is to get the best possible estimate of *Y* based on *X* values. In this scenario, the blue line fits the data much better than all the other lines provided. The black line is the mean of the *Y* value, which in this case is 3.9. Other lines (orange and gray) are not a good fit. Well, how did we come to this conclusion? We used the squared error algorithm to calculate the error for each line and chose the one that gave us the lowest error. These steps are taken to identify a simple linear regression model. Since we looked for a line with the least squared error, this is also called an ordinary least squares or *least squares regression*. By default, PySpark uses *squared error* as the loss function. Linear regression can be represented with the following equation:

$$y = \beta_0 + \beta_1 * x_1 + \beta_2 * x_2 + \ldots\ldots\ldots + \beta_n * x_n$$

where β_0 is the intercept and n is the number of variables in the linear regression model. Let's fit a linear regression model using PySpark and analyze the output. The target is the balance variable and the inputs are the rest of the continuous variables: age, day, duration, campaign, pdays, and previous.

```
#select the variables
linear_df = data.select(['age', 'balance', 'day', 'duration', 'campaign',
'pdays', 'previous'])
target_variable_name = 'balance'
#exclude target variable and select all other feature vectors
features_list = linear_df.columns
#features_list = char_vars #this option is used only for ChiSqselector
features_list.remove(target_variable_name)
# apply the function on our DataFrame
df = assemble_vectors(linear_df, features_list, target_variable_name)
# fit the regression model
from pyspark.ml.regression import LinearRegression
reg = LinearRegression(featuresCol='features', labelCol='balance')
reg_model = reg.fit(df) # fit model
# view the coefficients and intercepts for each variable
import pandas as pd
for k, v in df.schema["features"].metadata["ml_attr"]["attrs"].items():
    features_df = pd.DataFrame(v)
# print coefficient and intercept
print(reg_model.coefficients, reg_model.intercept)
features_df['coefficients'] = reg_model.coefficients
# prediction result
pred_result = reg_model.transform(df)
```

The regression coefficients are provided in Figure 5-16 (features_df dataset).

	idx	name	coefficients
0	0	age	28.083973
1	1	day	3.305546
2	2	duration	0.248828
3	3	campaign	-14.142676
4	4	pdays	-0.082488
5	5	previous	23.462993

Figure 5-16. *Regression coefficients*

To represent this model in an equation, you would write the following:

```
Predicted_balance = 124.92 + 28.08 * age + 3.30 * day +
           0.25 * duration + -14.14 * campaign
          -0.08 * pdays + 23.46 * previous
```

Interpreting the Linear Regression Model

Let's look at the significance of the age variable in the model. For this purpose, we take some example values for each input and substitute them into the preceding equation.

```
{Age:33, day:5, duration:76, campaign:1, pdays:-1, previous:0}
Predicted_balance = 124.92 + 28.08 * 33 + 3.30 * 5 +
           0.25 * 76 + -14.14 * 1
          -0.08 * -1 + 23.46 * 0 = 1073
```

That's perfect. Now, we increase the age value to 34 and keep the rest constant. Doing the same calculation again, we get

```
Predicted_balance = 1101.8
Difference in prediction due to age increase = 1101.8 - 1073 = 28.08
```

Now you can see the impact of age. In layman's terms, when age increases by 1, the predicted balance increases by 28 (rounded value) and vice versa. This is easily translatable into business logic, and decisions can be made based on this value. You can similarly interpret other input variables. The main things to consider while interpreting a linear regression model are the following:

1) Keep all other variables constant except the variable you are trying to interpret.

2) Pick the coefficient of that input variable and use it for interpretation in layman's terms.

Perfect. However, there is a slight problem here, discussed next.

Multicollinearity

While interpreting a regression model, you keep all the variables constant except one variable. What if the variable that you are trying to interpret is correlated with another input variable in the dataset? For example, assume that `age` and `duration` are correlated in the data. In this case, when you increase the `age` variable, the `duration` value also increases. You cannot keep the `duration` variable constant in this case. Therefore, the interpretability is lost in the process. This problem is called *multicollinearity*.

When two or more variables are correlated, the model results are not interpretable. In a linear regression model, you should avoid multicollinearity issues. Otherwise, the model output is not valid.

You can check for multicollinearity using the Variance Inflation Factor (VIF). The industry standard for VIF's value is 10. You can tweak this value to fit your needs. When the VIF value is 1, the input features are completely uncorrelated. PCA components usually have a VIF close to 1. In the final model, you should include input variables with a VIF value of less than 10. The formula to calculate VIF is provided here:

$$VIF = \frac{1}{1 - R^2}$$

The code for VIF in PySpark is provided here:

```
def vif_calculator(df, features_list):
    vif_list = []
    for i in features_list:
```

```
        temp_features_list = features_list.copy()
        temp_features_list.remove(i)
        temp_target = i
        assembler = VectorAssembler(inputCols=temp_features_list,
        outputCol='features')
        temp_df = assembler.transform(df)
        reg = LinearRegression(featuresCol='features', labelCol=i)
        reg_model = reg.fit(temp_df) # fit model
        temp_vif = 1/(1 - reg_model.summary.r2)
        vif_list.append(temp_vif)
    return vif_list
features_df['vif'] = vif_calculator(linear_df, features_list)
print(features_df)
```

Note `linear_df` *is provided instead of* `df`*. Make sure the dataset does not already have a column named* `features`*. Otherwise, this function will throw an error.*

The VIF for all the inputs is less than ten (see Figure 5-17). Therefore, there are no multicollinearity issues in the model features. When you want to find the best subset of features that produce the best linear regression model, you need to look at Mallows' Cp factor. This concept is outside the scope of this book. The next chapter examines different ways to validate a regression model and pick the best one.

	idx	name	coefficients	vif
0	0	age	28.083973	1.000917
1	1	day	3.305546	1.034350
2	2	duration	0.248828	1.007627
3	3	campaign	-14.142676	1.039907
4	4	pdays	-0.082488	1.276182
5	5	previous	23.462993	1.261321

Figure 5-17. *VIF*

Logistic Regression

Logistic regression is a classification algorithm. You use this algorithm to predict binary outcomes such as whether a person will buy something or not, churn versus no churn, spam versus no spam, and so on. Like linear regression, it fits a line on the data to separate the binary outcomes. However, the best-fitting line is decided based on the Maximum Likelihood Estimate. First, let's look at the equation for a logistic regression model:

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 * x_1 + \beta_2 * x_2 + \ldots\ldots\ldots + \beta_n * x_n$$

The equation for logistic regression differs from that for linear regression on the left component. This component, $\log\left(\frac{p}{1-p}\right)$, is called *log(odds)*. Why *log(odds)*? Look at Figure 5-18.

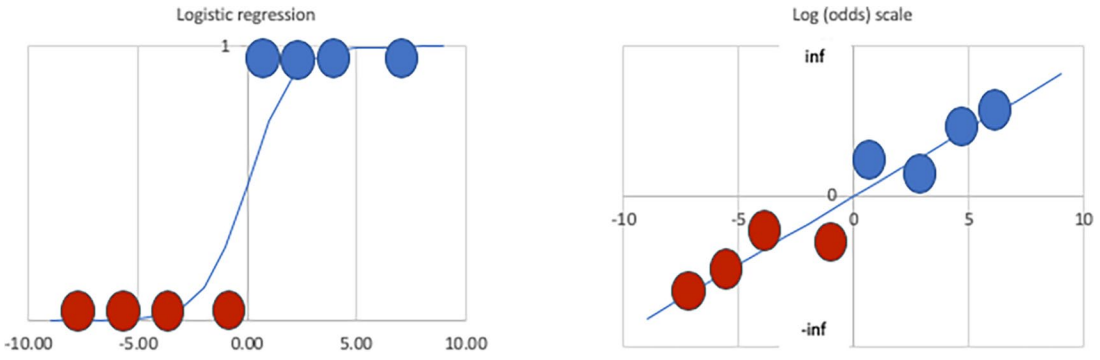


Figure 5-18. *Logistic regression scale versus log odds scale*

Figure 5-18 on the left shows the data you want to fit the regression line. By converting the 0/1 scale to a *log(odds)* scale, you can map the data in the range (-infinity, + infinity). In this scale, the logistic regression fits the best line.

```
Log(odds) positive range = log(1/0) = log(1) – log(0) = +Infinity
Log(odds) negative range = log(0/1) = log(0) = -Infinity
Log(odds = 0.5) = Log(0.5/0.5) = log(1) = 0
```

Therefore, 0.5 probability is the midpoint. As mentioned, the best fit is determined based on the Maximum Likelihood Estimation. So, what is it?

Maximum Likelihood Estimation or Maximum Log-likelihood (MLE)

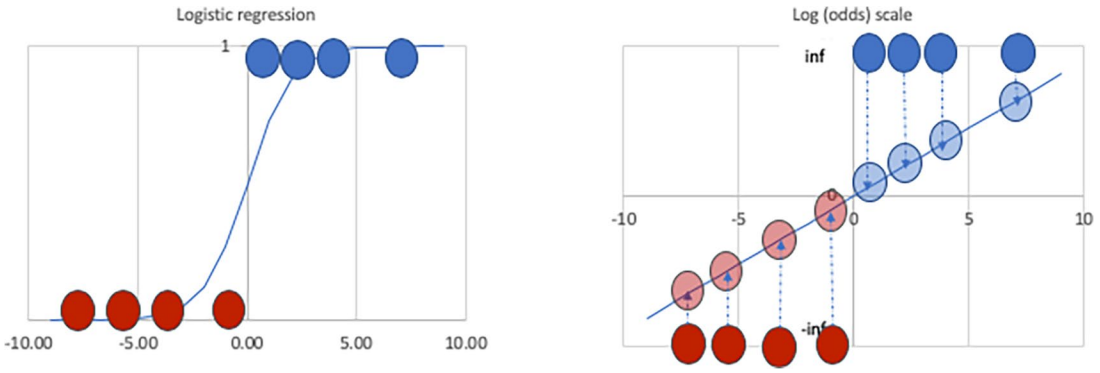


Figure 5-19. *Maximum Likelihood Estimation*

You pick a candidate line as shown on the right of Figure 5-19 and project your data points onto that line. In this process, you end up getting a `log(odds)` value for each data point based on the line. You convert the `log(odds)` to a probability scale so that you can convert the fitted line to a Sigmoid function similar to the one shown on the left. This is done using the following equation:

$$\log\left(\frac{p}{1-p}\right) = \log(odds)$$

This means

$$\frac{p}{1-p} = e^{\log(odds)}$$

Solving this equation, you get

$$p = \frac{e^{\log(odds)}}{1+e^{\log(odds)}} = \frac{1}{1+e^{-\log(odds)}} = \frac{1}{1+e^{-x}} = sigmoid\ \ function$$

where, $x = log(odds)$. You have the probability values for each observation. Let's start calculating the MLE for this line.

Note During MLE calculation, use *p* when the target is 1 and 1- *p* when the target is 0.

`Likelihood = prob (1st observation) * prob (2nd observation) * ..... prob (nth observation)`

where *n* is the total number of observations in your data and *prob* is the probability. Taking log on both sides, you get the following:

`Log(likelihood) = log (prob 1st) + log (prob 2nd) + ..... + log (prob nth)`

This is the equation you try to maximize in logistic regression. Hence, you call this process maximum log-likelihood estimation. You repeat this process by rotating the `log(odds)` line and calculating the `log(likelihood)` each time. Finally, the line that maximizes the `log(likelihood)` outcome is the best-fitting line.

How Do You Rotate the Line to Find the Best Fit?

Solvers can be used to rotate the line in a direction that maximizes the MLE value and reduces the number of iterations required to get a proper fit. In PySpark, the default solver is `lbfgs`, also known as Limited-Memory Broyden Fletcher Goldfarb Shanno.

Binary Versus Multinomial Classification

The Sigmoid function is given by this formula:

$$p(x) = \frac{e^{\log(odds)}}{1 + e^{\log(odds)}} = \frac{e^{x}}{e^{0} + e^{x}}$$

Here, the e^0 corresponds to the baseline model, which is the 0 class in a binary outcome. e^x corresponds to the 1 class. In a similar way, you can apply the same formula to a multi-class problem. Let's say you have classes *a*, *b*, and *c*. The probability of each class is given by

$$p(a) = \frac{e^{\log(odds)}}{1 + e^{\log(odds)}} = \frac{e^{a}}{e^{a} + e^{b} + e^{c}}$$

For other classes, the corresponding *e* element goes to the numerator. This is the SoftMax function, as discussed earlier. Now, you know the link between Sigmoid and SoftMax. Another thing to consider for binary versus multi-class is a link function. In general, you use the logit ($\log\left(\frac{p}{1-p}\right)$) link function for binary targets. For multinomial, you can use *probit* for nominal targets and *clog-log* for ordinal targets. In PySpark, you can accomplish a binary or multinomial fit by setting the *family* option in the logistic regression algorithm.

PySpark Code

For this exercise, the target variable is `y` in the bank dataset and the input predictors are `age`, `balance`, `day`, `duration`, `campaign`, `pdays`, and `previous`. The target `y` is a binary class.

```
+---+-----+
|  y|count|
+---+-----+
|0.0|39922|
|1.0| 5289|
+---+-----+
```

```
target_variable_name = "y"
logistic_df = data.select(['age', 'balance', 'day', 'duration', 'campaign',
'pdays', 'previous', 'y'])
#exclude target variable and select all other feature vectors
features_list = logistic_df.columns
#features_list = char_vars #this option is used only for ChiSqselector
features_list.remove(target_variable_name)
# apply the function on our dataframe
df = assemble_vectors(logistic_df, features_list, target_variable_name)
# using String indexer. To get around the issue, we had to perform
this step
indexer = StringIndexer(inputCol='y', outputCol='y_index',
handleInvalid='skip')
indexerModel = indexer.fit(df)
# Transform the DataFrame using the fitted StringIndexer model
df = indexerModel.transform(df)
# drop existing "y" column so we can rename new "y_index" column to "y"
df = df.drop(df.y)
# rename new "y_index" column to "y"
df = df.withColumnRenamed('y_index', 'y')
```

We showcase both binary and multinomial fit using the same target, although it is preferable to use multinomial fit when the target has more than two classes.

```
# Example showing binary and multinomial fit.
# Prefer multinomial fit when the target has more than two classes.
import numpy as np
from pyspark.ml.classification import LogisticRegression
```

```
binary_clf = LogisticRegression(featuresCol='features', labelCol='y',
family='binomial')
multinomial_clf = LogisticRegression(featuresCol='features', labelCol='y',
family='multinomial')
binary_clf_model = binary_clf.fit(df) # fit binary model
multinomial_clf_model = multinomial_clf.fit(df) # fit multinomial model
np.set_printoptions(precision=3, suppress=True)
#model coefficients for binary model
print(binary_clf_model.coefficients)
#model coefficients for multinomial model
np.set_printoptions(precision=4, suppress=True)
print(multinomial_clf_model.coefficientMatrix)
print(binary_clf_model.intercept) #model intercept for binary model
#model intercept for multinomial model
print(multinomial_clf_model.interceptVector)
```

The equation for the binary model is provided here:

```
log(odds class1) = -3.47 + 0.008*age + 0*balance -0.0017*day +
0.0036*duration - 0.128*campaign + 0.0021*pdays + 0.0859*previous
```

Similarly, the equation for the multinomial model is provided here:

```
log(odds class0) = 1.735 -0.004*age – 0*balance + 0.0008*day -
0.0018*duration + 0.064*campaign - 0.0011*pdays - 0.043*previous
```

and

```
log(odds class1) = -1.735 +0.004*age + 0*balance - 0.0008*day +
0.0018*duration - 0.064*campaign + 0.0011*pdays + 0.043*previous
```

In the multinomial model output, when you subtract the class 1 coefficient from the class 0 coefficient, you get the binary model coefficient for class 1. The same thing applies to the intercept too. Okay, now let's interpret the model results.

Interpreting the Model Results

You follow the same process used for interpreting the linear regression coefficient—keeping all the variables constant and changing the variable you are trying to interpret.

Binary Variable Interpretation

When it comes to binary variables, you calculate the `log(odds)` for both classes using the preceding equation. For example, assume you had gender (Male = 0, Female = 1) in your model. Then, the impact of gender is measured by the following:

```
log(odds class 1|Male) = Intercept + 0.16 * 0 + coef2 * var2 + ......
log(odds class 1|Female) = Intercept + 0.16 * 1 + coef2 * var2 + ......
log(odds class 1|Female) - log(odds class 1|Male) = 0.16
```

By keeping all the other factors constant, you get the coefficient of the binary variable as the difference in `log(odds)`. The best way to interpret the coefficient is by using odds. Let's do that.

$$\log\left(\frac{\text{odds class 1}|\text{Female}}{\text{odds class 1}|\text{Male}}\right)=0.16$$

$$\frac{\text{odds class 1}|\text{Female}}{\text{odds class 1}|\text{Male}}=e^{0.16}=1.173$$

You take the difference between the preceding value and 1 to interpret the odds. The odds of a female being in class 1 is 17 percent higher than the odds of a male being in class 1.

Continuous Variable Interpretation

Let's perform the same calculation with age.

```
log(odds class 1|Age = 29) = Intercept - 0.32 * 29 + coef2 * var2 + ......
log(odds class 1|Age = 30) = Intercept – 0.32 * 30 + coef2 * var2 + ......
log(odds class 1| Age = 30) - log(odds class 1| Age = 29) = -0.32
```

$$\log\left(\frac{\text{odds class 1}|\text{Age}=30}{\text{odds class 1}|\text{Age}=29}\right)=-0.32$$

$$\frac{\text{odds class 1}|\text{Age}=30}{\text{odds class 1}|\text{Age}=29}=\text{e}^{-0.32}=0.73$$

Now, 1 – 0.73 = .27. When age increases by one unit, the odds of a person being in class 1 decreases by 27 percent.

Here is the rule of thumb for interpreting the odds ratio:

1) An odds ratio of 1 means there is no difference in impact.
2) An odds ratio of greater than 1 means the odds increase as the variable changes.
3) An odds ratio of less than 1 means the odds decrease as the variable changes.

Decision Trees

So far, we have discussed linear methods for regression and classification. This section discusses non-linear methods that can be used to solve a problem. Overall, the idea behind decision trees is to generate a set of "if-then rules" and represent them in a tree. Let's visualize a simple decision tree for credit card approval.

Interpretation of the Decision Tree

- The root/parent node of the tree is the age variable. Therefore, it is the most important feature for this decision tree.
- The leaf nodes are the decisions that can be taken based on the input variables.

Let's traverse the tree for these conditions (see Figure 5-20).

Age = 20, occupation = Student, Credit score = 800	**Age = 35, occupation = Employed, Credit score = 750**
Age < 25; True Student? True	Age < 25; False Age < 60; True Credit Score > 720
Final verdict = Not approved	Final verdict = Approved

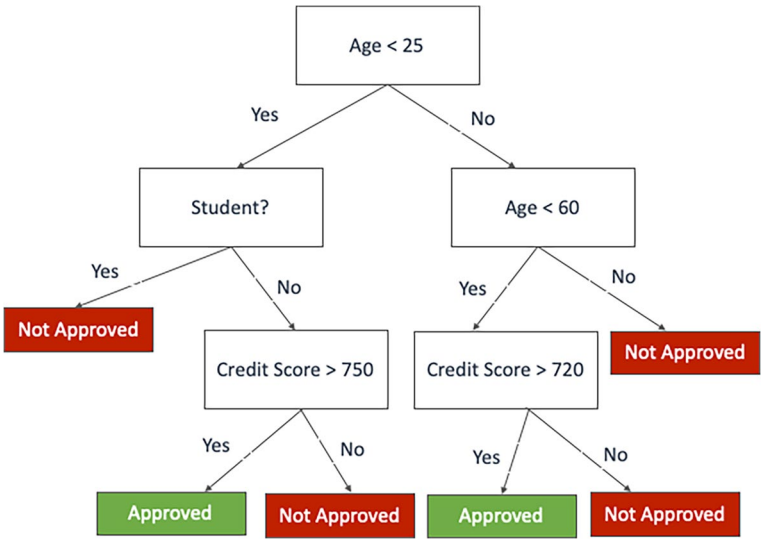


Figure 5-20. *Credit card approval decision tree*

You can see that age is the most important attribute. How did the decision tree pick this variable? In order to understand why, we need to cover Gini, entropy, and information gain. Let's review each of them (see Table 5-3).

Table 5-3. *Algorithm Specifications*

Algorithm/Split Criterion	Type of Tree
Gini impurity	CART (Classification and Regression Trees)
Entropy and information gain	ID3/C4.5 (Iterative Dichotomiser3/ C4.5 is an extensive to ID3 and uses gain ratio)

For this exercise, we use the dataset shown in Table 5-4. The data target is credit card approval with two levels: approved and not approved.

Table 5-4. *Dataset for Exercise*

Age	Gender	Approved/Not Approved
30	Male	Not approved
25	Female	Approved
45	Male	Approved
57	Female	Not approved
27	Male	Approved
54	Female	Approved
35	Female	Not approved

Entropy

Entropy is the measure of randomness in the data. The formula to calculate entropy is

$$Entropy = -\sum p(x) \log_2 p(x)$$

It ranges between 0 and 1. Entropy is 0 when there is no randomness, and entropy is 1 when the dataset is completely random. To understand entropy, consider the following example.

You have two bags. In the first bag, there are four blue balls and no red balls. In the second bag, there are two blue balls and two red balls. Calculate the entropy in both bags.

Bag 1 Entropy

```
Probability of picking blue ball in bag 1 = 4/4 = 1
Probability of picking red ball in bag 1 = 0/4 = 0
Entropy = - (1 * log(1) + 0* log(0)) = 0
```

Bag 2 Entropy

```
Probability of picking blue ball in bag 2 = 2/4 = 0.5
Probability of picking red ball in bag 2 = 2/4 = 0.5
Entropy = - (0.5 * log(0.5) + 0.5* log(0.5)) = 1
```

Let's extend this example and give you a choice to pick a bag. You win the game when you pick a blue ball. Which bag would you choose?

The answer is obvious, right? You pick the first bag because it has all the blue balls. When the entropy is 0, you are certain of the outcome, whereas when the entropy is 1, the outcome is random. This methodology is used in machine learning to build the decision tree.

Let's switch back to the example data and calculate the entropy for age and gender. This time we calculate the probability of being approved. For the purpose of this demonstration, let's split the age variable into two parts—above 30 and below 30.

Entropy of Target (Approved/Not Approved) – Parent Entropy:

```
Probability of approved = 4/7 = 0.57
Probability of not approved = 3/7 = 0.43
Entropy of parent node = - (0.57 * log(0.57) + 0.43 * log(0.43)) = 0.99
```

Entropy of Age Variable:

```
Probability of approved when age >= 30 = 2/5 = 0.4
Probability of not approved when age >= 30 = 3/5 = 0.6
Probability of approved when age < 20 = 2/2 = 1
Probability of not approved when age < 20 = 0/2 = 0
Entropy(Age >= 30) = - (0.4 * log(0.4) + 0.6* log(0.6)) = 0.97
Entropy(Age < 30) = - (1 * log(1) + 0 * log(0)) = 0
```

Entropy of Gender Variable:

```
Probability of approved when female = 2/4 = 0.5
Probability of not approved when female = 2/4 = 0.5
Probability of approved when male = 2/3 = 0.67
Probability of not approved when male = 1/3 = 0.33
Entropy(Gender = Female) = - (0.5 * log(0.5) + 0.5* log(0.5)) = 1
Entropy(Gender = Male) = - (0.67 * log(0.67) + 0.33 * log(0.33)) = 0.92
```

Information Gain

Up until now, we have calculated the entropy for each variable. Now it's time to determine the variable that goes to the root/parent node. To make this decision, we use information gain. Information gain tells you the knowledge gained in each split. You are

trying to maximize your knowledge on credit card approval. Therefore, you will try to maximize the information gain in each split. It is calculated using the following formula:

```
Information Gain, IG = Entropy (parent node) – (Entropy each child node *
proportion of observations in each child node)
```

Now calculate the information of each variable. Since five observations went in age >= 30 and two observations in age < 30, the information gain is given by

```
IG (age >= 30)  = 0.99 – (0.97 * 5/7 + 0 * 2/7) = 0.29
```

In a similar way, the IG for gender is given by

```
IG gender = 0.99 – (1 * 4/7 + 0.92 * 3/7) = 0.02
```

The age variable has higher information gain than the gender variable. Therefore, we pick age as the root node. We can start building the tree based on this condition (see Figure 5-21).

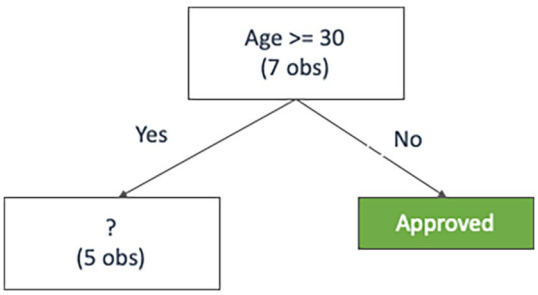


Figure 5-21. *Information gain first split*

Based on this current model, when a person's age is less than 30, the credit card is approved. Our job now is to figure out the next split. We will iterate through the same process again. For age, we will use 45 as the split point this time.

```
Entropy (Age >=45) = - (0.67 * log(0.67) + 0.33 * log(0.33)) = 0.92
Entropy (Age < 45) = - (0 * log(0) + 1* log(1)) = 0
Entropy (Gender = Female) = -(0.33*log(0.33) + 0.67 * log(0.67))= 0.92
Entropy (Gender = Male) = - (0.5 * log(0.5) + 0.5* log(0.5)) = 1
IG (age >= 45)  = 0.97 – (0.92 * 3/5 + 0 * 2/5) = 0.418
IG (Gender) = 0.97 – (0.92 * 3/5 + 1 * 2/5) = 0.018
```

Again, information gain for age is higher than for the gender variable. So, the next split would look like Figure 5-22.

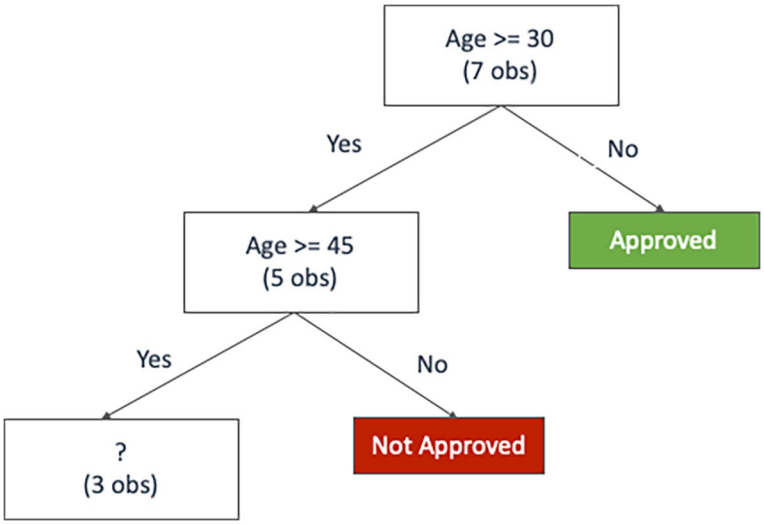


Figure 5-22. *Information gain second split*

For the next split, we use age as 60 and repeat the process.

```
IG (age >= 60) = 0.92 - (0 * 0 + 0.92) = 0
IG (Gender) = 0.92 - (0 * 1/3 + 1 * 2/3) = 0.25
```

After this split, there are only two observations, and since both are female, we cannot use gender anymore. Therefore, we split based on age, and we get the final tree, shown in Figure 5-23.

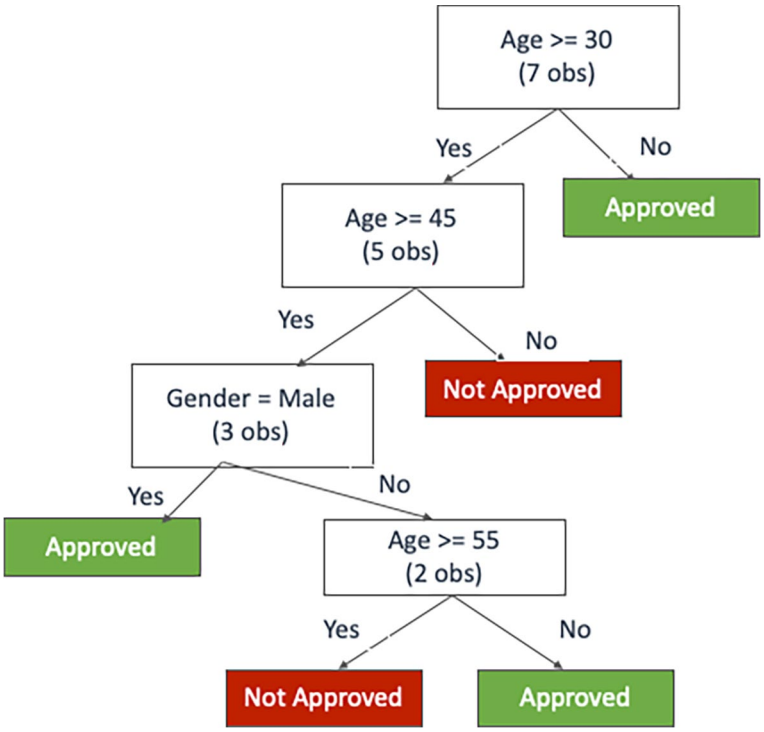


Figure 5-23. *Final tree based on information gain*

That is all. You run through the decision tree to get the output when you get new data. ID3, C4.5, and C5.0 decision trees are built using entropy/information gain.

Gini Impurity

Let's look at another approach to forming a decision tree, Gini impurity. You should not confuse this concept with the Gini coefficient. Gini impurity measures the impurity in a split. The formula for Gini impurity is as follows:

$$Gini = 1 - \sum \left(p(x) \right)^2$$

The range of the Gini impurity is between 0 and 1.

- When 0, the split is pure and has no impurities.
- When 0.5, the split is random and completely impure. Both classes are equally present in the data.

- Any value above 0.5 can occur when one of the child nodes does not have a single data point. In the example dataset, when you consider Age >=60, the Gini impurity for this split will be more than 0.5 since you don't have any data points with an age greater than 60.

Let's calculate the Gini impurity for the red/blue ball example to understand this concept.

Gini Impurity – Bag 1

```
Probability of picking blue ball in bag 1 = 4/4 = 1
Probability of picking red ball in bag 1 = 0/4 = 0
Gini = 1 - (1 * 1 + 0 * 0) = 0
```

Gini Impurity – Bag 2

```
Probability of picking blue ball in bag 2 = 2/4 = 0.5
Probability of picking red ball in bag 2 = 2/4 = 0.5
Gini = 1 - (0.5 * 0.5 + 0.5 * 0.5) = 0.5
```

Gini Impurity – Bag Is Empty

```
Probability of picking blue ball in empty bag = 0
Probability of picking red ball in empty bag = 0
Gini = 1 - 0 = 1
```

Again, when you need to pick a bag, you will pick the one with a Gini impurity of 0 because you are certain of the outcome. Let's apply the Gini concept to the data to build a decision tree. We follow the same steps used for entropy and calculate the impurity at each split.

```
Gini (Age >= 30) = 1 - (2/5 * 2/5 + 3/5 * 3/5) = 0.48
Gini (Age < 30) = 1 - (2/2 * 2/2 + 0/2 * 0/2) = 0
Weighted Gini for Age > = 30 = (5/7 * 0.48 + 2/7 * 0) = 0.34
Gini (Gender = Female) = 1 – (2/4 * 2/4 + 2/4 * 2/4) = 0.5
Gini (Gender = Male) = 1 = (1/3 * 1/3 + 2/3 * 2/3) = 0.44
Weighted Gini for Gender = (4/7 * 0.5 + 3/7 * 0.44) = 0.48
```

The age variable is used for the first split in the tree. Let's continue this process and calculate the final tree. You will get the exact same tree as you do in entropy and information gain. This is because of the choice of the bins during the age split. However,

this is not always the case. In real-world examples, the Gini impurity tree will differ from the entropy and information gain tree. Classification and Regression Trees (CART) are built using Gini impurity.

Variance

One more method that PySpark uses to create decision trees is variance. The formula for variance is

$$Variance = \frac{\sum\left(X - mean\left(X\right)\right)^2}{n}$$

This concept is used to build regression trees. You can continue splitting the tree by choosing the variable that minimizes the variance at each split. Decision trees can also be made using other criteria, like chi-square and correlation. The concept is the same. You pick variables that maximize the chi-square value in a chi-square. For a correlation tree, you pick variables that minimize the correlation. These concepts are out of the scope of this book.

Determine the Cut-off for Numerical Variables

We used a custom cut-off for age while building the decision tree by hand. In a decision tree algorithm, numeric variables are handled in a different way. For small datasets, each unique value is used as a cut-off point. For large datasets, the numeric variables are binned into a specific number of quantiles. This is set using the `maxBins` option in the decision tree algorithm. After creating the quantiles, the average value of each bin is used as a cut-off point. For example, assume you choose three bins for the age variable, and your cut-off points after binning are 25, 42, and 55. The information gain or Gini impurity is calculated at each of these cut-off points, and the point that provides the best outcome is chosen for the split. The same process is repeated for all numeric variables at each split.

Pruning the Tree/Stopping Criterion

Pruning a tree refers to chopping off the unwanted leaf nodes. This is done to keep the model simple and to avoid overfitting. We discuss overfitting in detail in the next chapter. Decision trees are prone to overfitting. Unless you control this, you will end up having a tree that memorizes training data. Pruning can be accomplished in a tree using the following options:

- `maxDepth` – The depth of a tree refers to the number of tree traversal steps required from a root node to reach the leaf node. By default, the maximum depth of a tree is 5.
- `minInstancesPerNode` – The minimum number of samples to be present in each child node is specified using this option. When you set this option to 5 and a split produces a child node with fewer than five observations, the split will not happen.
- `minInfoGain` – A tree split should provide information gain of at least this number specified. When the information gain is less than this number, the split will not happen.
- `L1/L2 Regularization` – This option can be used to prune decision trees. We discuss this in later sections.

PySpark Code for Decision Tree Classification

The dataset used in the linear and the logistic regression examples is renamed to `continuous_df` and `binary_df`, respectively.

```
from pyspark.ml.classification import DecisionTreeClassifier
clf = DecisionTreeClassifier(featuresCol='features', labelCol='y',
impurity='gini') #gini based model
clf_model = clf.fit(binary_df)
clf2 = DecisionTreeClassifier(featuresCol='features', labelCol='y',
impurity='entropy') #entropy based model
clf_model2 = clf2.fit(binary_df)
clf_model.transform(binary_df) #future predictions
# gini feature importance
print(clf_model.featureImportances)
#output - (7,[0,2,3,4,5],[0.057,0.001,0.722,0.0006, 0.217])
print(clf_model2.featureImportances)
#output - (7,[0,2,3,4,5],[0.018,0.001,0.727,0.0004,0.254])
```

As you can see from the preceding output, the Gini and entropy methods produce a similar set of variables; however, in the real world, the set of variables and trees will be different for each method.

PySpark Code for Decision Tree Regression

To fit a regression tree, you must set the impurity as a variance. By default, decision tree regression uses the variance method. Unfortunately, PySpark does not support this.

```
from pyspark.ml.regression import DecisionTreeRegressor
# Define the decisiontreeregressor with variance impurity
reg = DecisionTreeRegressor(featuresCol='features', labelCol='balance',
impurity='variance')
reg_model = reg.fit(continuous_df)
print(reg_model.featureImportances) #feature importance
reg_model.transform(continuous_df) #future predictions
```

Similar to classification trees, you can interpret the feature importance of the input variables. To look at the if-then rules of the decision tree, you can use the following code:

```
clf_model.toDebugString
reg_mode.toDebugString
```

You can plot these decision tree rules using the `Graphviz`, `pydot`, or `networkx` packages in Python.

Feature Importance Using Decision Trees

In the previous chapter, we used decision trees to calculate feature importance. In this section, we go through how this is calculated.

$$f_i = \sum_{j:nodes\ splits\ on\ feature\ i} s_j * G_j$$

where

$$f_i = \text{importance of feature}$$

$$s_j = \text{number of samples in node } j$$

$$G_j = \text{impurity value of node } j$$

At each node, we calculate the Gini impurity and multiply it by the number of samples that went through the node. If a variable appears more than once in a decision tree, the outputs of each node are added to get the final feature importance for that variable. Finally, the feature importance of each variable is normalized so that they all add to 1. That is done using the following equation. Let's say you have *n features;* the final feature importance is given by

$$\text{final} - f_i = \frac{f_i}{f_1 + f_2 + \ldots\ldots + f_n}$$

Where

$f_1, f_2, \ldots \supset, f_n$ are the individual feature importance of each variable

f_i – feature importance of the i^{th} variable calculated using the previous equation

$\text{final} - f_i$ – normalized feature importance of variable i

You need to be aware that the feature importance is calculated based on the training dataset. Therefore, the feature importance could be biased when the model is overfitting. To overcome this, you can use proper validation techniques or use permutation importance to calculate the feature importance. We talk about validation techniques in the next chapter. The permutation importance concept is out of the scope of this book. Let's now move on to other tree-based techniques.

Random Forests

In a decision tree, you build a single tree based on the training data. Instead of a single tree, what if you build multiple trees and aggregate the results of each tree to get the final prediction? That is the idea behind random forests. A *random forest* is an ensemble model built on a concept called bagging.

- *Ensemble* means a collection of different models. The individual models can be built using a variety of machine learning algorithms like logistic regression, decision tree, neural networks, and so on. The final result is determined based on voting (in the case of classification) or the average of predictions (in the case of regression).

- *Bagging*, a.k.a bootstrap aggregation, is a concept by which random subsets of data (both rows and columns) are used to train a model. In a random forest algorithm, multiple decision trees are trained on different subsets of the same dataset. For example, the first tree might train using `age` and `gender` as input variables, the second tree might train using `balance` and `income` as input variables, and so on. In addition, the rows (samples) used to train the first and second tree are randomly sampled from the original training dataset. You can sample rows *with or without replacement*. When you choose to go *with replacement*, the same sample can be used more than once. In a *without replacement* scenario, the same sample cannot be used more than once.

Random forests use both these concepts to build trees. The process is as follows:

1. Select random features – subset features (`featureSubsetStrategy`).
2. Select random samples (with or without replacement) – subset rows (`subsamplingRate`).
3. Form the subset data based on the subset features and rows.
4. Build a decision tree based on the subset data.
5. Repeat Steps 1-4 to build the number of trees specified by the user. The PySpark default for the number of trees is 20 (`numTrees`).
6. For the final prediction using random forest, run the data through all the individual trees. Get the prediction of each tree.
7. For classification, count the number of votes for each class. The class with the maximum number of votes is chosen as the final output.
8. For regression, average the output (predictions) of all individual trees to get the final output.

Hyperparameter Tuning

By tuning the `featureSubsetStrategy` and `subsamplingRate` parameters, the model performance and training process speed can both be improved.

`featureSubsetStrategy - "auto", "all", "sqrt", "log2", "onethird"`

`subsamplingRate` - any value between 0 and 1. When the value is set to 1, the whole dataset is used.

Feature Importance Using Random Forest

In random forests, feature importance is calculated in a similar way as for a decision tree, with one slight change. The average of the feature importance of a variable is calculated before the normalization step.

$$\text{average} - f_i = \frac{f_i}{\text{number of trees with feature } f_i}$$

The $average - f_i$ for all variables is used to calculate the final feature importance, $final - f_i$.

The same concept is used in gradient-boosting trees as well, which you see next.

PySpark Code for Random Forest

Classification

```
from pyspark.ml.classification import RandomForestClassifier
clf = RandomForestClassifier(featuresCol='features', labelCol='y')
clf_model = clf.fit(binary_df)
print(clf_model.featureImportances)
print(clf_model.toDebugString)
```

Regression

```
from pyspark.ml.regression import RandomForestRegressor
reg = RandomForestRegressor(featuresCol='features', labelCol='balance')
reg_model = reg.fit(continuous_df)
print(reg_model.featureImportances)
print(reg_model.toDebugString)
```

Why Random Forest?

So far, we have not covered why we need random forests. Let's go through that now.

- Random forests are more robust than a single decision tree, and they limit overfitting.
- They remove feature selection bias by randomly selecting features during each tree training process.
- Random forest use a proximity matrix, which can be used to fill in missing values.

Gradient Boosting

Another variant of the decision tree is gradient-boosted trees. Gradient boosting is an ensemble model built on the boosting concept. We discussed the ensemble concept. Let's look at boosting in more detail.

Boosting uses weak learners to build trees. It is an additive modeling technique. It is also referred to as *sequential learning* because the current tree learns from the mistakes of the previous tree.

Boosting Learning Process

- Model training starts by building an initial decision tree. Predictions are made using this tree.
- A sample weightage column is created based on the prediction output. When there is an error in prediction, the sample is given more weightage, and when the predictions are accurate, the sample is given less weightage.
- You create a new training dataset based on the sample weightage column. This is to ensure that the error samples are given more preference in the newly created training data.

- You repeat the training process with this new data and continue building trees until the user-specified `numTrees` option is met.
- Learning rate (`learningRate`) is used to scale each weak learner output.
- The final model prediction is given by the following:

```
First tree prediction + learningRate*Second tree prediction + ..... +
learningRate * Nth tree prediction
```

PySpark Code for Gradient Boosting

Classification

```
from pyspark.ml.classification import GBTClassifier
clf = GBTClassifier(featuresCol='features', labelCol='y')
clf_model = clf.fit(binary_df)
print(clf_model.featureImportances)
print(clf_model.toDebugString)
```

Regression

```
from pyspark.ml.regression import GBTRegressor
reg = GBTRegressor(featuresCol='features', labelCol='balance')
reg_model = reg.fit(continuous_df)
print(reg_model.featureImportances)
print(reg_model.toDebugString)
```

Why Gradient Boosting?

- Useful for modeling imbalanced target classes.
- Gradient boosting builds shallow trees, as compared to the deep trees created by random forest and decision trees. Therefore, it is good at reducing bias in predictions.

Support Vector Machine (SVM)

Imagine you are in a classroom full of students. Your task is to draw a line that separates the boys from the girls. In addition to drawing the line, your method should also have these characteristics:

- A good margin of separation between boys and girls
- A low error rate during separation

This is accomplished using support vector machines (SVM). In general, support vector machines use hyperplanes to perform classification. Look at Figure 5-24 to see the concept in detail.

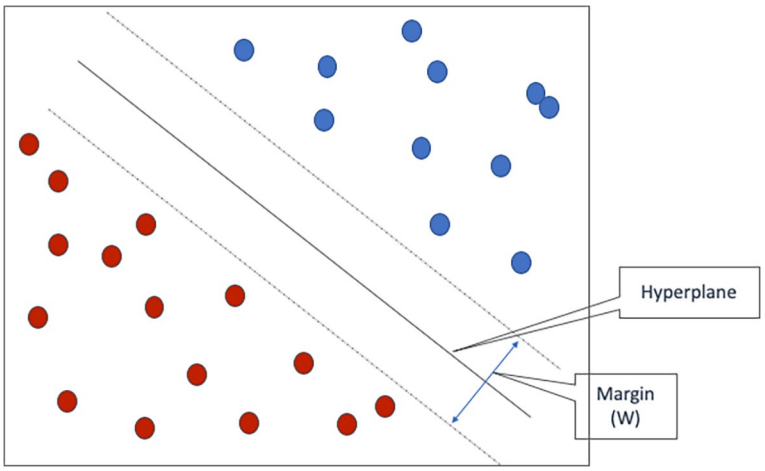


Figure 5-24. *Feature space separation using a support vector machine*

In the figure, a linear hyperplane is used to separate the classes. For non-linear data, you can use polynomial or rbf (radial basis function) hyperplanes to get a good separation. This is called a kernel trick. Currently, PySpark supports only the linear kernel. However, Python version supports other kernels. The SVM model comprises two error functions, as follows (see Figure 5-25):

```
Total error = Classification error + Margin error
```

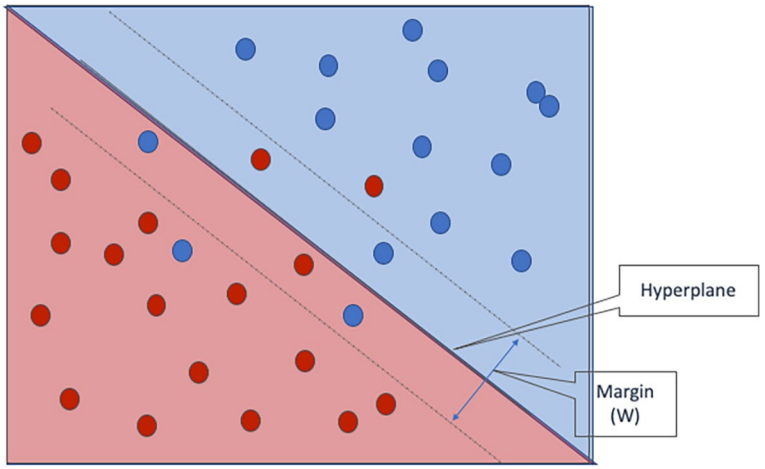


Figure 5-25. *Classification and margin errors*

Classification Error

Classification error measures the error in prediction in the SVM model. This error is similar to misclassification rate. However, we mentioned that the margin is also a component of SVM. Therefore, any observation that falls within the margin is also considered an error since you are trying to maximize the margin. In Figure 5-25, there are seven samples that contribute to classification error: two outside the margin and five within the margin. The total classification error is the sum of the absolute value of the distances of these observations from the margin boundaries. In order to calculate the red observation error, you calculate distance from the upper margin, and for the blue observation error, you calculate distance from the lower margin.

Margin Error

Margin error quantifies the error associated with the margin boundaries. When you have a large margin, the error is small and vice versa. Let's represent the line in the form of an equation, as follows:

$$ax + by + c = - - - > center\ \ line$$

$$ax + by + c = i - - - > upper\ \ margin$$

$$ax + by + c = -i - - - > lower\ \ margin$$

You can calculate the width of the margin using the following formula:

$$W = \frac{2i}{\sqrt{a^2 + b^2}}$$

And the margin error is given by the following:

$$Margin\ error = a^2 + b^2$$

When you substitute values in the preceding equation, you get a small error for a large margin and a large error for a small margin.

Total Error

As you saw before, total error is the sum of the classification and margin errors. In general, a large margin produces a large classification error and vice versa. You can tweak the model error function by using a regularization parameter (`regParam`). Later sections focus on regularization.

PySpark Code

```
from pyspark.ml.classification import LinearSVC
# Define LinearSVC classifier with features and label columns
clf = LinearSVC(featuresCol='features', labelCol='y')
clf_model = clf.fit(binary_df)
print(binary_df.intercept, clf_model.coefficients)
```

Note SVM takes a long time to train. You can play with the `regParam` option to train the model faster.

Neural Networks

Neural networks are inspired by the biological neurons in the human brain. In humans, brains communicate with other parts of the body using these neurons. Depending upon the task, the brain activates the neurons corresponding to the task and accomplishes it.

Artificial neural networks (ANNs) are built on this concept and use activation functions to activate certain neurons and get the required output. A simple neural network architecture is shown in Figure 5-26.

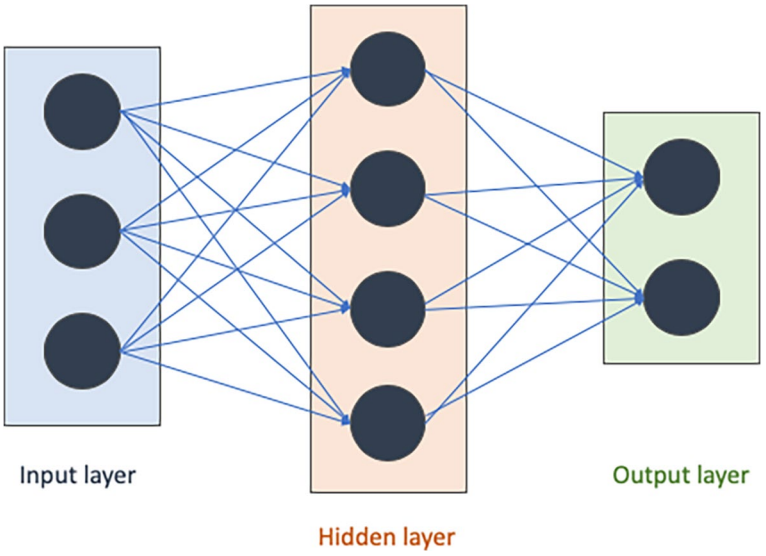


Figure 5-26. *ANN architecture*

Things to Know About ANN Architecture

- The simplest neural networks include input and output layers without a hidden layer. This type of network is called a *perceptron* (see Figure 5-27). A linear or logistic regression model can be built with a perceptron neural network.

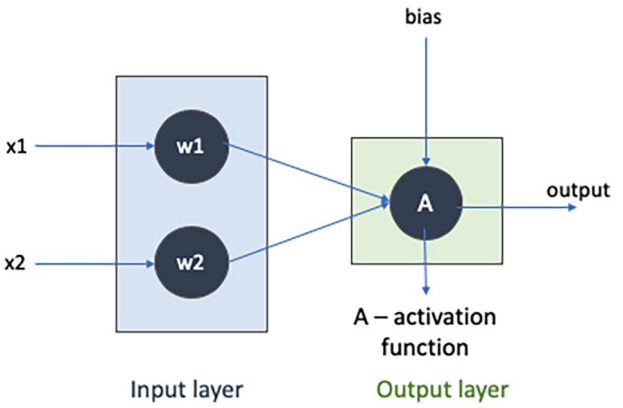


Figure 5-27. *Perceptron*

- When the activation function is linear, the perceptron produces a linear regression, and when the activation function is Sigmoid, the perceptron produces a logistic regression.
- You can add multiple hidden layers between input and output layers in a perceptron. Because of this capability, neural networks are also called multi-layer perceptrons.
- A single unit in a hidden layer is called a hidden unit. In Figure 5-26, there are four hidden units in the hidden layer.
- You can use activation functions to turn on/off a hidden unit. When you use a linear activation function, it models a linear relationship. By using non-linear activation functions like ReLu and TanH, you can model a complex relationship.
- When you stack multiple hidden layers, you produce a deep neural network.
- When you increase the size of the hidden units in a hidden layer, you produce a wider neural network.

- Neural networks do not need prior knowledge about the data. They can model the relationship between features and targets using these two steps: feed-forward (Step 1), backpropagation and gradient descent (Step 2). This is why neural networks are also called feed-forward neural networks.
- In PySpark, neural networks can be optimized using hyperparameters such as `maxIter`, `layers`, `blockSize`, and `stepSize`

How Does a Neural Network Fit the Data?

We use logic gates to explain the concept in detail. For this example, we use an OR gate. For this exercise, we use a perceptron network. The data is listed in Table 5-5.

Table 5-5. *Sample Data*

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	1

After a neural network architecture is set up, the bias and weights are randomly initialized. In general, a gaussian or uniform function is used to initialize bias and weights. We will set the bias as 1 for this example. Since there are two features in this example, we will use two weights (w1 and w2). Let's assume the initial random weight values are w1 – 0.1 and w2 – 0.7. The activation function for this example is Sigmoid.

$$\text{Sigmoid function, } \sigma = \frac{1}{1+\exp(-x)}$$

$$final\ output = \begin{cases} 0\ if\ \sigma < 0.5 \\ 1\ if\ \sigma \geq 0.5 \end{cases}$$

Neural network model output = $w1 * x1 + w2 * x2 + b$

Step 1: Feed-Forward

In a feed-forward step, the weights are multiplied by the inputs and sent to the output function to calculate the final output (see Table 5-6).

Table 5-6. *Error Calculations*

Input1	Input2	Target	Output = 0.1 * Input1 + 0.7 * Input2 + 1	Sigmoid(output) = $\frac{1}{1+\exp(-output)}$	Error = Target - Sigmoid (output)
0	0	0	1	0.73	-0.73
0	1	1	1.7	0.85	0.15
1	0	1	1.1	0.75	0.25
1	1	1	1.8	0.86	0.14

Step 2: Backpropagation

Let's backpropagate the errors to update the weights of the input features. To update the weights, use the following formula:

```
New weight = old weight -( learning rate) * error delta
Error delta = input * error * derivative(output)
```

We set the learning rate as 1. As you can see from the preceding equation, we take the derivate of each error output to update the weight (see Table 5-7). The derivative of the Sigmoid function is given by the following:

$$\text{Derivation of Sigmoid function}, \sigma' = \sigma * (1 - \sigma)$$

Table 5-7. *Derivative Calculations*

Error	I = Derivative(sigmoid)	J = Error * I	W1_Delta = Input1 * J	W2_Delta = Input2 * J
-0.73	0.19	-0.14	0	0
0.15	0.13	0.02	0	0.02
0.25	0.18	0.05	0.05	0
0.14	0.12	0.02	0.02	0.02

```
W1_errorDelta = sum(W1_Delta) = 0.07
W2_errorDelta = sum(W2_Delta) = 0.04
New_w1 = 0.1 - 0.07 = 0.03
New_w2 = 0.7 - 0.04 = 0.66
```

That's it. The first iteration is (epoch) complete, and the weights are updated after one epoch. You simply repeat the steps for a certain number of epochs to train your model. During each step, the weight gets updated, and slowly the model learns to fit according to the data.

Hyperparameters Tuning in Neural Networks

- `maxIter` – The number of epochs to run. One epoch includes both feed-forward and backpropagation. When the number of epochs is high, the model takes more time to train, and vice versa. In addition, when the number of epochs is very low, the model will not be able to learn the patterns in the data. On the other hand, a larger epoch makes the model more prone to overfitting.
- `layers` – The number of hidden layers to use while training the model.
- `Format – [hidden layer1, ...., Hidden layer` *n*`, output layer]`. When the hidden layer increases, you can get more accurate predictions. However, the model is prone to overfit the data. When the layer decreases, the model won't have enough space to learn. The trick is to find the right balance. A good number to start with is *n* or 2*n*, where *n* is the number of input features.

- `blockSize` – The batch size. The number of batches to be used while training the model. This is the mini-batch concept discussed previously.
- `stepSize` – The learning rate for gradient descent. How fast should your model learn? We discussed this in detail in the previous section.

PySpark Code

```
from pyspark.ml.classification import MultilayerPerceptronClassifier
#output_layer is set to 2 because of binary target
clf = MultilayerPerceptronClassifier(featuresCol='features', labelCol='y',
layers=[4, 4, 2])
clf_model = clf.fit(binary_df)
```

Currently, PySpark does not support regression using neural networks. However, you can implement regression using a linear activation function.

One-vs-Rest Classifier

This is used for multi-class classification exercises. The idea behind this classifier is very simple. Let's say you have four classes for `education`—`primary`, `secondary`, `tertiary`, and `unknown`. Your objective is to predict the `education` outcome for an individual. This is a multi-class problem because you have more than two output classes. You create a base classifier and use it to perform binary classification for each class. The first classifier will predict primary versus non-primary (`secondary`, `tertiary`, or `unknown`). The second classifier will predict secondary versus non-secondary (`primary`, `tertiary`, or `unknown`). In a similar fashion, the third classifier will predict tertiary versus nontertiary, and the fourth classifier will predict unknown versus non-unknown. The final prediction is obtained by evaluating these four individual classifiers and choosing the index of the most confident classifier as output.

PySpark Code

```
# Multi-class classification model using a RandomForestClassifier
target_variable_name = "education"
multiclass_df = data.select(['age', 'balance', 'day', 'duration',
'campaign', 'pdays', 'previous', 'job', 'education'])
features_list = multiclass_df.columns
#features_list = char_vars #this option is used only for ChiSqselector
features_list.remove(target_variable_name)
# apply the function on our dataframe
# Assemble features into a vector column
multiclass_df = assemble_vectors(multiclass_df, features_list, target_
variable_name)
# fitting the one-vs-rest classifier
from pyspark.ml.classification import RandomForestClassifier, OneVsRest
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
# generate the train/test split.
(train, test) = multiclass_df.randomSplit([0.7, 0.3])
# instantiate the base classifier.
clf = RandomForestClassifier(featuresCol='features', labelCol='education')
# instantiate the One Vs Rest Classifier.
ovr = OneVsRest(classifier=clf, featuresCol='features',
labelCol='education')
# train the multiclass model.
ovrModel = ovr.fit(train)
# score the model on test data.
predictions = ovrModel.transform(test)
# obtain evaluator.
evaluator = MulticlassClassificationEvaluator(metricName="accuracy",
labelCol='education')
# compute the classification error on test data.
accuracy = evaluator.evaluate(predictions)
print("Test Error = %g" % (1.0 - accuracy))
```

Naïve Bayes Classifier

The Naïve Bayes classifier is based on the Bayes theorem. It assumes the independence of the input variables. This is why it is called *naïve*. Input variables are not always independent of each other. Bayes' theorem calculates posterior probability based on prior probability and likelihood. It is denoted by the following:

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$

where,
A & B are the events.
P(A|B) = Posterior probability. It is read as "probability of A given B is true"
P(B|A) = Likelihood. It is read as "probability of B given A is true"
P(A) = Prior probability of event A
P(B) = Prior probability of event B

Look at the data in Table 5-8 to understand the Bayes theorem. We have two buckets, each with ten balls. The first bucket has seven red balls and three green balls, and the second has seven green balls and three red balls. Assume you have an equal chance of choosing the buckets. Assume you randomly took a ball from one of the buckets, which came out as green. What is the probability that the ball came from bucket 2?

Table 5-8. *Bayes Demonstration—Sample Data*

Bucket1	Bucket2
Red	Red
Red	Green
Red	Green
Green	Green
Red	Green
Red	Green
Green	Red
Red	Red
Red	Green
Green	Green

There are multiple ways to find the answer to this question. We use the Bayes theorem for this problem since it is most suitable for these cases.

- You have two events: A – *Bucket*2 and *B* – green color ball.
- You have equal chances of selecting buckets, so the prior probability of *Bucket* 2 = P(A) = 0.5
- The prior probability of choosing a green ball (GB) which is event B is given by P(B)
- P(B) = P(*Bucket*1) * P(GB in *Bucket*1) + P(*Bucket*2) * P(GB in *Bucket*2)
- P(B) = 0.5 * 0.3 + 0.5 * 0.7 = 0.5
- The likelihood of a green ball given *Bucket*2 is given by

$$P(Greenball \mid Bucket2) = P(B|A) = 0.7$$

- Probability of *Bucket*2 given green ball:

$$P(\text{Bucket2}|\text{Greenball}) = P(A|B) = \frac{P(B|A)*P(A)}{P(B)} = (0.7 * 0.5)/0.5 = 0.7$$

Bayes' theorem provides a simple solution for these types of problems. The classifier is an extension of this theorem. The Naïve Bayes classifier is useful for training models on large datasets because of its simplicity. It is most suitable for categorical input features. For numerical input features, you can manually create categorical variables using WOE or other encoding techniques. You can also feed numerical variables without transformation, but the algorithm assumes that the distribution of the variable is normal (you have to be cautious about this assumption).

One flip side of this classifier is the *zero frequency* problem. When you have a new category in test/score data that is not present in training data, the model will output the probability as 0. In these situations, you can use a smoothing technique. PySpark supports additive smoothing (also called *Laplace smoothing*) to smooth categorical data. The default value for smoothing is 1. When the value is set to 0, no smoothing will be performed. Another disadvantage of using this classifier is that it expects non-negative feature values. When your dataset is completely numeric and has negative values, it is better to standardize your data before using this classifier or to choose another classifier. The Naïve Bayes classifier is more suitable for classifying documents such as spam detection or multi-class classification.

PySpark Code

```
target_variable_name = "y"
nonneg_df = data.select(['age', 'day', 'duration', 'campaign',
'previous', 'y'])
#exclude target variable and select all other feature vectors
features_list = nonneg_df.columns
#features_list = char_vars #this option is used only for ChiSqselector
features_list.remove(target_variable_name)
# apply the function on our DataFrame
nonneg_df = assemble_vectors(nonneg_df, features_list, target_
variable_name)
# fit Naïve Bayes model
from pyspark.ml.classification import NaiveBayes
clf = NaiveBayes(featuresCol='features', labelCol='y')
clf_model = clf.fit(nonneg_df)
```

Regularization

Regularization is a technique used to avoid overfitting. In general, there are three types of regularization

- L1 Regularization or Lasso
- L2 Regularization or Ridge
- Elastic-net, a combination of L1 and L2

Parameters

`elasticNetParam` is denoted by *alpha*. `regParam` is denoted by *lambda*. The regularization term is provided here:

$$\alpha\left(\ \lambda\ w_1\right)+\left(1-\alpha\right)\left(\frac{\lambda}{2}\ w_2^2\right),\alpha\in\left[0,1\right],\lambda\geq 0$$

The first component in the equation is the L1 or Lasso term, and the second component is the L2 or Ridge term. When `elasticNetParam` is set to 1, it performs Lasso regularization, and when it is set to 0, it performs Ridge regularization. When the value is somewhere in between 0 and 1, it performs elastic-net regularization.

Lasso or L1 Penalty

- Lasso penalty takes the absolute value of the slope and performs regularization.
- Lasso excludes variables that are not useful for the model. This can be used for feature selection.

Ridge or L2 Penalty

- Ridge penalty takes the squared value of the slope and performs regularization.
- It diminishes the coefficients of variables that are less useful for the model, aiding in feature selection.

These techniques are useful when you are trying to evaluate a bunch of hyperparameters to select the best hyperparameters and to avoid overfitting.

Summary

- You learned about supervised techniques.
- You learned about various error functions and activation functions.
- You looked look at the different optimizers available in the ML field.
- You explored a variety of supervised algorithms and saw use cases for regression and classification.
- Finally, you learned about regularization and how to use it to avoid overfitting.

Great job! In the next chapter, you learn about model validation. Keep learning and stay tuned.

CHAPTER 6

Model Evaluation

"All models are wrong, but some are useful."

— George E.P. Box.

Many people try to develop models to perform a certain task (for example, predicting house prices). Oftentimes, these models cannot represent 100 percent of reality. In the real estate example, we cannot exactly predict a house price all the time. However, it does not mean that the model is garbage. In general, all statistical and machine learning models face this problem. Then, why build one in the first place? Even though we cannot represent reality 100 percent, we can still model useful behavior and represent reality closely enough. For example, we can use demographic information like ZIP code to predict price, and this model can perform better than randomness. This fact lays the foundation for this entire chapter. You need to ensure that this idea is planted well because everybody cares about the insight you bring to the table based on your model.

That being said, you can build one or more models to predict house prices. Which one do you pick, and why? This chapter provides tips and tricks for answering these questions. But before we dive into the model evaluation metrics, we need to cover some related concepts.

Model Complexity

Model complexity measures how well your model generalizes the information it has learned from training. In the previous chapter, we split the data into training and test data. Then, we built our model on the training dataset (model fit) and applied the fitted model to the test dataset (model generalization). The whole process is shown

© Ramcharan Kakarla, Sundar Krishnan, Balaji Dhamodharan and Venkata Gunnu 2024
R. Kakarla et al., *Applied Data Science Using PySpark*, https://doi.org/10.1007/979-8-8688-0820-3_6

in Figure 6-1. As an example, we used three classification models. The same concept applies when you compare models generated using different hyperparameters and several epochs of the same algorithm—like in neural networks.

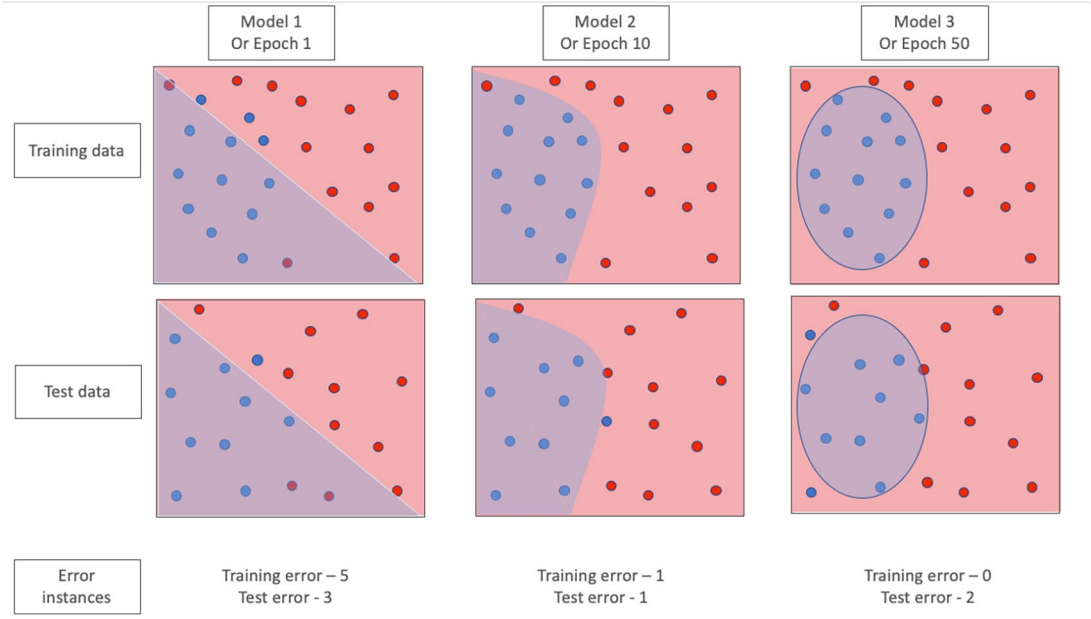


Figure 6-1. *Comparing performance of models with different settings*

Underfitting

Look at Model 1 in Figure 6-1. It is a linear model that does not fit the training data well. Therefore, it does not generalize well on the test data either. The model fails to capture the non-linearity in the underlying data. As a result, the error level in the training and the test datasets is high. This is called *underfitting*. When this type of issue happens, the model performs poorly on the training and test data.

Best Fitting

Now look at the Model 2 in Figure 6-1. It captures the non-linearity in the data better. As a result, the model performs equally well with the training and the test data. Both models fit, and model generalization is good. This is a best-fitting-model scenario. As a data scientist or machine learning enthusiast, you should always look for this sweet spot—the Goldilocks spot.

Overfitting

Now look at Model 3 in Figure 6-1. This model fits the training data perfectly. However, it fails to generalize on the test data well. This is called *overfitting*. You can also overfit a model by setting a larger number of epochs during the training process. As a result, the model memorizes (instead of generalizes) the training data well and fails to perform equally well on the test data.

When you plot the training and testing errors for each model, you typically get the graph shown in Figure 6-2. This figure is generated based on the number of error instances in Figure 6-1.

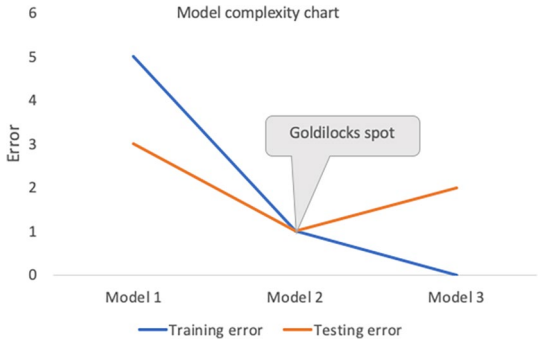


Figure 6-2. *Model complexity chart*

Oftentimes, you'll want to see errors in terms of a percentage of the whole dataset—that is, the misclassification rate. Let's generate another figure with the misclassification rate, and this time we will use the error level at each epoch instead of using different models. You can clearly see the difference in the misclassification rates as a result of underfitting, best fit, and overfitting. By looking at this chart, you can pick the best model for your data, which is the Goldilocks spot, as shown in Figure 6-3. At Epoch 10, the model performs better on the training and test data, as shown in Figure 6-3.

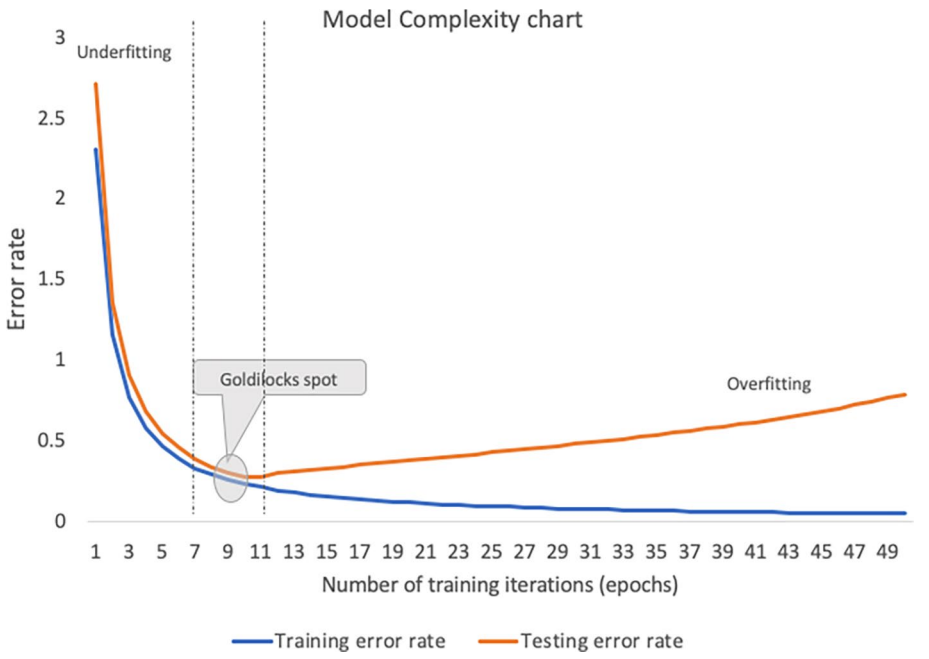


Figure 6-3. *Goldilocks spot*

Bias and Variance

When discussing model complexity, some people use the terms *bias* and *variance* to describe it. *In general, an ideal model should have low bias and low variance.* So, what are they?

- *Bias* tells you the ability of an algorithm to represent the true relationship in the data.
- *Variance* tells you the consistency of an algorithm (performance) across different datasets.

In the current example, Model 1 is unable to capture the non-linear relationship in the data, so it has high bias. However, it produces similar errors in the training and test datasets, so it has low variance. In a similar fashion, Model 2 has low bias and low variance, and Model 3 has low bias and high variance. As a result, we selected Model 2 as the best model.

Model Validation

You should now understand the purpose of having a training/test split during modeling based on what we discussed about model complexity. Having training and testing datasets is one way to perform model validation. There are other methods available as well. Let's start with the training/test split and then go on to other validation techniques.

Training/Test Split

The idea is simple, as you have seen all along. You randomly split your data according to a specified size. In most of the examples, we use a 70/30 split. This means that 70 percent of the data is roughly sampled from the original dataset to form the training set, and the remaining 30 percent goes to the test set. This is shown in Figure 6-4.

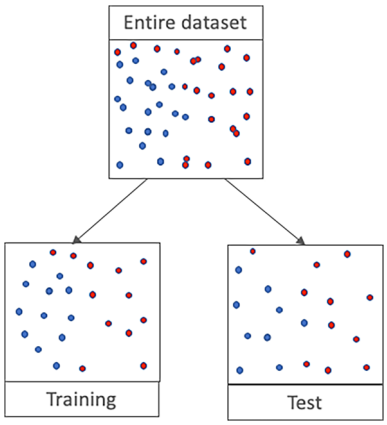


Figure 6-4. *Training/test split*

Another way to represent this method is by using the bars shown in Figure 6-5.

Figure 6-5. *Training/test split*

When performing the split, you can use simple random sampling or stratified sampling.

Simple Random Sampling

Data points are assigned at random to each training and test dataset according to the proportion specified by the user. For a 70/30 split, 70 percent of the data is randomly chosen to create the training dataset, and the remaining 30 percent is assigned to the test dataset.

Stratified Sampling

This type of sampling is done in supervised studies. Look at the distribution of the target in the original dataset. Let's assume that the distribution is 60 percent zeros and 40 percent ones. With stratified sampling, even after the split, you still maintain the 60/40 distribution of targets in both the training and test datasets.

Other sampling methods—such as equal sampling, cluster sampling, and systematic sampling—are outside the scope of this book. In PySpark, you can accomplish a training/test split using one of three options. The following code performs a 70/30 split.

Option 1

```
train, test = df.randomSplit([0.7, 0.3], seed=12345)
```

Option 2

The second option uses the PySpark *ML tuning* library. Before this step, you need to use the vector assembler output as the input.

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit
#model initialization
lr = LogisticRegression(maxIter=10, featuresCol='features',
labelCol='label')
#model parameters to try - parameter grid setup
paramGrid = ParamGridBuilder().addGrid(lr.regParam, [0.1, 0.01]).
addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]).build()
# 70% of the data will be used for training, 30% for validation.
train_valid_clf = TrainValidationSplit(estimator=lr,
estimatorParamMaps=paramGrid, evaluator=BinaryClassificationEvaluator(),
```

```
trainRatio=0.7)
# assembled_df is the output of the vector assembler
model = train_valid_clf.fit(assembled_df)
```

Option 3

To perform stratified sampling in PySpark, you need to use a workaround using the `randomSplit` option. The code is shown here:

```
# split data for 0s and 1s
zero_df = df.filter(df["label"]==0)
one_df = df.filter(df["label"]==1)
# split data into train and test
train_zero, test_zero = zero_df.randomSplit([0.7,0.3], seed=12345)
train_one, test_one = one_df.randomSplit([0.7,0.3], seed=12345)
# union datasets
train = train_zero.union(train_one)
test = test_zero.union(test_one)
```

Sampling Bias

You have seen the benefits of using a training/test split. However, there are some drawbacks to this approach. Imagine you are predicting some retail sales and are using November's data. Due to Thanksgiving, the store advertised a huge discount, and sales increased by *x* percent. When you use this data to build a model and predict sales for the rest of the month, you might get a model that does not account for the holiday discount. This is called a *sampling bias*.

Sampling bias happens when your data is skewed toward one segment of the population and does not account for all segments in the population. As a modeler, you need to understand the data before building your model. You should also understand the circumstances under which your model can be used. Otherwise, the model will not produce expected results. Sampling bias could be minimized by using one of these techniques: holdout/out-of-time or cross-validation.

Holdout/Out-of-Time

The holdout concept extends the idea of the training/test split, except that this time you have three datasets instead of two. See Figure 6-6.

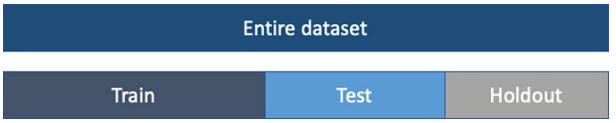


Figure 6-6. *Holdout concept*

To create a holdout dataset in PySpark, you might have to make a slight modification to the code that you saw before. This will produce a 70/20/10 split.

```
train, test, holdout = df.randomSplit([0.7, 0.2, 0.1], seed=12345)
```

In addition, you can have training and test datasets before you execute Option 2 and fit the model on the training dataset instead of on the entire dataset. This way, the test dataset can become the holdout dataset.

The holdout dataset partially solves the sampling bias issue, but the data still comes from the same timeframe. To get around this, you need to consider out-of-time data. In the retail sales prediction example, November would be training/test/holdout, and then some other month (for example, June) would be out-of-time data. *Industry practice recommends you have at least two out-of-time data samples to validate your model.* This way, you can account for the holiday discount's significance in your model and make your predictions stable.

Population Stability Index

Another recommendation for avoiding sampling bias and achieving a stable model is to perform a population stability index (PSI) check on the variables across datasets (training/test/holdout/out-of-time). Essentially, you are measuring the distribution changes in each variable with respect to the target across your data. You pick the best variables with high feature importance and low variability (low PSI index). This way, your model produces stable predictions over time. The PSI check is a simple formula:

$$PSI = \sum\left(Actual\% - Expected\%\right) * ln\left(\frac{Actual\%}{Expected\%}\right)$$

This equation should resemble the Information Value equation you saw in the Weight of Evidence (WOE) section in Chapter 4.

$$IV = \sum(\text{Event\%} - \text{Non Event\%}) * ln\left(\frac{\text{Event\%}}{\text{NonEvent\%}}\right)$$

The only difference between an IV and a PSI is that the IV value compares the distribution of an event versus a non-event of the same variable, whereas the PSI compares the distribution of the event for a variable from two different time periods (for example, training versus out-of-time). Therefore, the training data event distribution is the *expected* outcome and the rest of the dataset's event distribution is considered the *actual* outcome. Variables with high IV values are preferred in IV analysis, while variables with low PSI values are favored in PSI analysis. The cut-off for PSI is slightly different from IV too. See Table 6-1.

Table 6-1. *PSI Values*

PSI Value	Inference	Action
Less than 0.1	Low variability	Use this variable
0.1 - 0.25	Medium variability	Perform some transformation to make the variable stable. Cautiously, use this variable.
Greater than 0.25	High Variability	Important to address the variable stability issue before using it in model. Consider alternative variables, if stability cannot be achieved.

Let's do a simple PSI calculation for the training and holdout datasets for a single variable. The results are shown in Table 6-2.

Note Run WOE code on the training dataset and apply the bins on the holdout dataset. The training dataset will always include the *expected* values.

Table 6-2. *Training vs Holdout Datasets*

varname	Value	Expected % (Train Event distribution)	Actual % (Holdout Event distribution)	Acutal % - Expected %	ln (Actual%/Expected%)	Index
month	jan	0.102822581	0.097323601	-0.00549898	-0.054963466	0.00030224
month	feb	0.166931638	0.165354331	-0.001577307	-0.009493742	1.4975E-05
month	mar	0.520833333	0.517730496	-0.003102837	-0.005975263	1.854E-05
month	apr	0.187592319	0.217536071	0.029943752	0.148093588	0.00443448
month	may	0.066604128	0.068552253	0.001948126	0.028829722	5.6164E-05
month	jun	0.100320171	0.106716886	0.006396716	0.061812627	0.0003954
month	jul	0.091519731	0.089629282	-0.001890449	-0.020872517	3.9458E-05
month	aug	0.114146793	0.101141079	-0.013005714	-0.120968918	0.00157329
month	sep	0.465517241	0.462427746	-0.003089496	-0.006658815	2.0572E-05
month	oct	0.431034483	0.453703704	0.022669221	0.051256257	0.00116194
month	nov	0.108976727	0.085510689	-0.023466038	-0.242492962	0.00569035
month	dec	0.456953642	0.492063492	0.03510985	0.07402581	0.00259904
					PSI	0.016306

The PSI value for the `month` variable is 0.016. It indicates that the variable is stable (low variability) between the training and holdout datasets. You can perform the same check for all the variables in your model and ensure they are stable across both datasets. As a side note, you can do PSI checks for model outputs and monitor the model's performance across time periods.

k-fold Cross-Validation

Until now, you have seen an approach of holdout/out-of-time that can handle sampling bias. What if you have to work with a single dataset and don't have an out-of-time dataset? This type of situation arises, especially when you deal with survey data. K-fold cross-validation can assist you in these situations. The idea is simple.

- Split the data randomly into k folds (typically, the value is 5 or 10, a configurable parameter).
- Fit the model on $k-1$ folds and test the model on the left-out fold (kth fold).
- Measure the metrics (accuracy/errors) for this model.
- Repeat the process each time until all the folds are used as the test dataset.
- The final metric (accuracy/errors) is the average of all the k-fold metrics.

A pictorial representation of the k-fold (four-fold) cross-validation is shown in Figure 6-7.

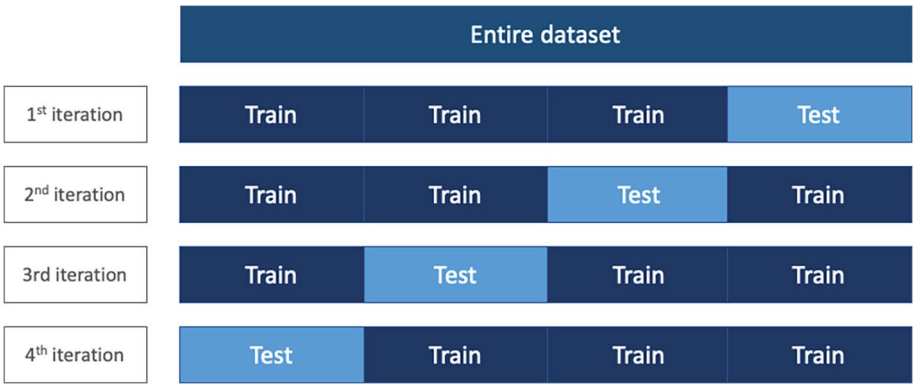


Figure 6-7. *Four-fold cross-validation*

In a k-fold cross-validation, every data point is used to train and test the model. Each data point can be either in training or test at a given time, but not both. Therefore, the training and test data are not static, and thus, the sampling bias is minimized.

In PySpark, you can perform k-fold cross-validation using the *ML tuning* library.

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
#model initialization
lr = LogisticRegression(maxIter=10, featuresCol='features',
labelCol='label')
#model parameters to try
paramGrid = ParamGridBuilder().addGrid(lr.regParam, [0.1, 0.01]).
addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]).build()
# number of folds = 3
crossval_clf = CrossValidator(estimator=lr, estimatorParamMaps=paramGrid,
evaluator=BinaryClassificationEvaluator(), numFolds=3)
# assembled_df is the output of the vector assembler
model = crossval_clf.fit(assembled_df)
```

The code does a three-fold cross-validation. Can you guess the number of models being trained with the preceding code? If your guess is 18, you are right. Notice the parameter grid used to train the cross-validation model. There are three parameters for *elasticNet*, two for *regularization parameters*, and three *folds* (3 * 2 * 3 = 18 models). As

you can see, cross-validation quickly becomes expensive when you increase the number of grid parameters or folds. However, this approach becomes useful for *hyperparameter tuning* instead of manually trying each parameter setting.

Leave-One-Out Cross-Validation

Leave-one-out cross-validation is a variation of k-fold cross-validation. The idea is to use a single observation as the test set and use the rest of the data as a training set. Figure 6-8 shows a pictorial representation of this method.

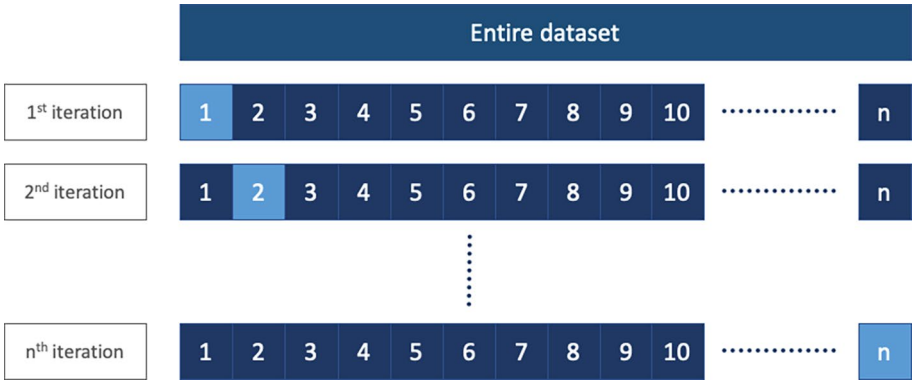


Figure 6-8. *Leave-one-out cross-validation*

Here, n is the number of observations in the data. In PySpark, you can perform leaveoneout cross-validation using the size of the dataset (n) as the number of folds in the k-fold cross-validation code shown previously. This method is computationally expensive, as the model needs to be trained n times for one parameter setting.

Leave-One-Group-Out Cross-Validation

This is another variant of k-fold cross-validation. It becomes useful when you want to test the model performance across different segments of people. Let's say you are doing a demographic study and want to see which age group will respond favorably to a marketing campaign. A k-fold method cannot be used here because it does not account for the age-group segments in each fold. You want to keep your age group clean while validating the model's performance. This is where leave-one-group-out cross-validation comes into play. A pictorial representation of this method is shown in Figure 6-9 with four groups.

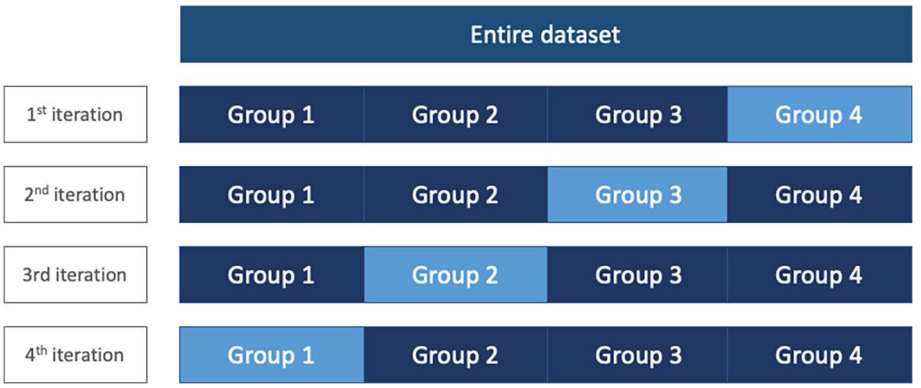


Figure 6-9. *Leave-one-group-out cross-validation*

This method is not currently available in PySpark. We show some workaround code here:

```
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler
from pyspark.ml import Pipeline
from pyspark.sql.functions import countDistinct
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.sql import functions as F
import numpy as np
filename = "bank-full.csv"
spark = SparkSession.builder.getOrCreate()
df = spark.read.csv(filename, header=True, inferSchema=True, sep=';')
df = df.withColumn('label', F.when(F.col("y") == 'yes', 1).otherwise(0))
df = df.drop('y')
df = df.select(['education', 'age', 'balance', 'day', 'duration',
'campaign', 'pdays', 'previous', 'label'])
features_list = ['age', 'balance', 'day', 'duration', 'campaign', 'pdays',
'previous']
#assemble individual columns to one column - 'features'
def assemble_vectors(df, features_list, target_variable_name, group_
variable_name):
    stages = []
```

```
    #assemble vectors
    assembler = VectorAssembler(inputCols=features_list,
    outputCol='features')
    stages = [assembler]
    #select all the columns + target + newly created 'features' column
    selectedCols = [group_variable_name, target_variable_name, 'features']
    #use pipeline to process sequentially
    pipeline = Pipeline(stages=stages)
    #assembler model
    assembleModel = pipeline.fit(df)
    #apply assembler model on data
    df = assembleModel.transform(df).select(selectedCols)
    return df
# apply the function on our DataFrame
joined_df = assemble_vectors(df, features_list, 'label', 'education')
# find the groups to apply cross validation
groups = list(joined_df.select('education').toPandas()['education'].
unique())
# leave-one-group-out validation
def leave_one_group_out_validator(df, var_name, groups):
    train_metric_score = []
    test_metric_score = []
    for i in groups:
        train = df.filter(df[var_name]!=i)
        test = df.filter(df[var_name]==i)
        #model initialization
        lr = LogisticRegression(maxIter=10, featuresCol='features',
        labelCol='label')
        evaluator = BinaryClassificationEvaluator(labelCol='label',
        rawPredictionCol='rawPrediction', metricName='areaUnderROC')
        #fit model
        lrModel = lr.fit(train)
        #make predicitons
        predict_train = lrModel.transform(train)
```

```
        predict_test = lrModel.transform(test)
        train_metric_score.append(evaluator.evaluate(predict_train))
        test_metric_score.append(evaluator.evaluate(predict_test))
        print(str(i) + " Group evaluation")
        print(" Train AUC - ", train_metric_score[-1])
        print(" Test AUC - ", test_metric_score[-1])
    print('Final evaluation for model')
    print('Train ROC', np.mean(train_metric_score))
    print('Test ROC', np.mean(test_metric_score))
```

Time-series Model Validation

Regarding time-series validation, the training/test split is done a little differently from what you have seen so far. Let's say you are working with monthly time-series data for the past year and want to perform a 70/30 split. You should pick the first eight months of data for training and the next four months for testing. K-fold crossvalidation could still be used here; however, the training data should always happen before the test data in all the folds.

Leakage

Another common problem in machine learning is *leakage*, which happens when you train a model using information that is not available at the time of prediction. Leakage can occur in the data and in the target. In either case, the model will not perform as expected when deployed in production.

Target Leakage

We use a simple example to explain this issue. Problem statement: "Predict the customers who are likely to buy a product in the upcoming month." You are using July's data, and ideally your target should come from August for this case. Instead, you create the target based on July. In this case, you are predicting the event after it happened. This does not solve the problem statement you defined and is an example of target leakage.

Another variant of target leakage is when you use an input feature to derive the target variable. Say you want to predict whether a house will sell above one million dollars. You derive the target from the sales price (above 1 million dollars – 1 and below 1 million dollars – 0). In this case, target leakage happens when you include the sales price variable in the model training, since it is a proxy for the target. When target definition is complex, this problem is more likely to occur.

Data Leakage

This type of leakage is more common than target leakage. Assume you have a 70/30 (training/test) split. Instead of training the model on the 70 percent training data, you train the model on the entire dataset (100 percent). In this situation, the model has already seen the test data. As a result, the model's evaluation metrics are biased.

Another variant of this issue occurs when you bootstrap the dataset before the training/test split. Bootstrapping is a concept by which you replicate the datapoints randomly. It is also known as *random sampling with replacement*. When you split the data after bootstrapping, it is more likely that the same datapoint is present in both datasets, thereby causing data leakage. In time-series data, when you split the data randomly instead of following the criteria (training happens before testing), it results in data leakage.

Issues with Leakage

Target or data leakage can result in significant issues, as follows:

1) The model cannot be deployed because the data is unavailable.
2) Model performance metrics are biased.
3) The model outputs are unstable, so the model fails to work in production.

Proper business understanding, feature engineering, evaluation of your data/target definitions with business stakeholders, and out-of-time datasets can help you resolve leakage issues.

Model Assessment

It's now time to dive deep into model assessment metrics. Depending on the type of model, PySpark offers a variety of metrics, as shown in Figure 6-10.

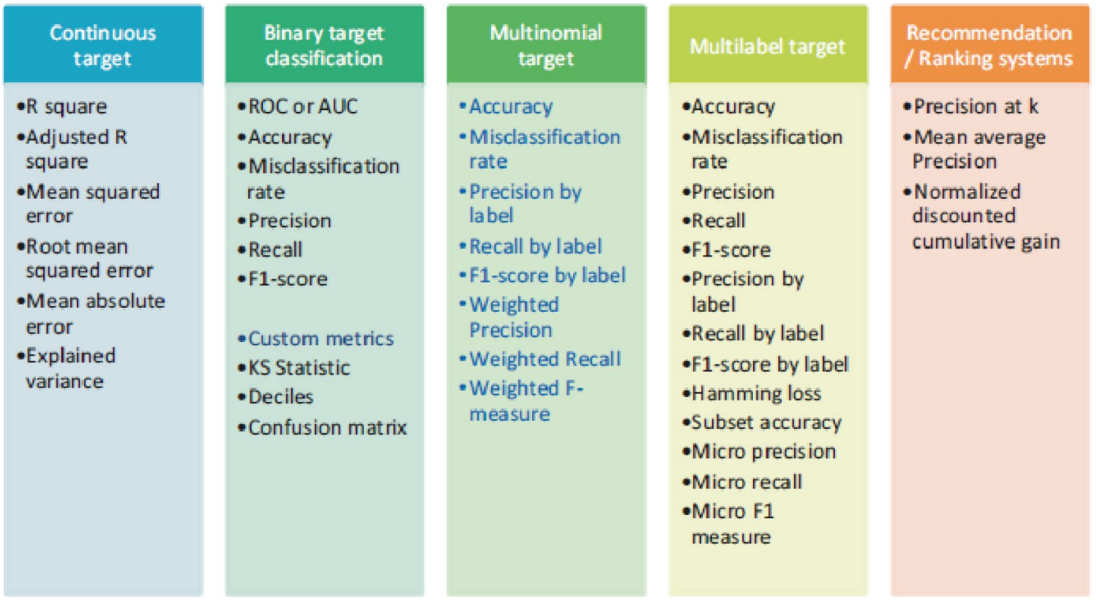


Figure 6-10. *Model assessment metrics*

Continuous Target

You can evaluate continuous targets using the `RegressionEvaluator` option in the PySpark *ML Tuning* package. Let's look at the sample data.

x1	x2	target
58	50	12
37	95	27
29	137	39
19	150	45

Figure 6-11 shows the data plot with the regression trendline (fit). We used Excel to produce the plot. You will get the same output using the PySpark `LinearRegression` model shown in the figure.

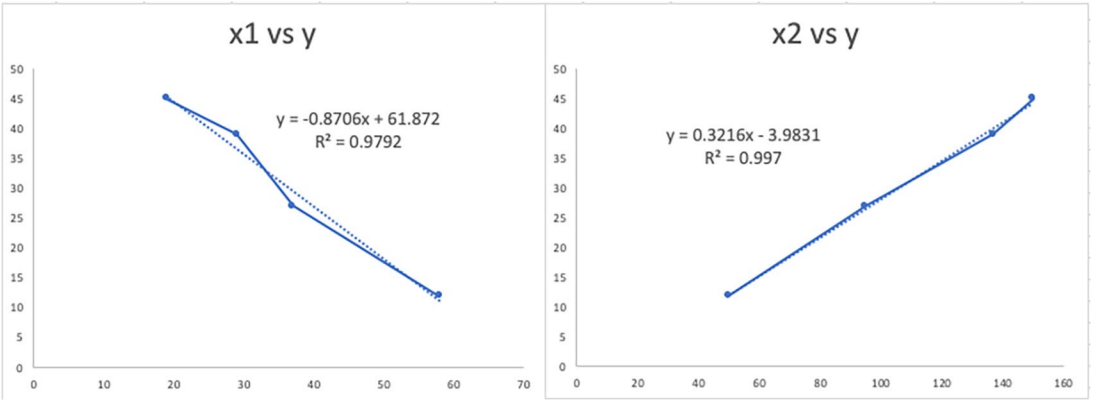


Figure 6-11. *Regression evaluation*

We use *x1 vs y* analysis to go through the entirety of the regression evaluation metrics.

Code to Replicate the Output Using PySpark

```
from pyspark.sql.types import IntegerType
from pyspark.sql.types import StructField, StructType
cSchema = StructType([StructField("x1", IntegerType())\
                      ,StructField("x2", IntegerType())\
                      ,StructField("y", IntegerType())])
df_list = [[58, 50, 12], [37, 95, 27], [29, 137, 39], [19, 150, 45]]
df = spark.createDataFrame(df_list, schema=cSchema)
# vector assembler we have used in earlier chapters
assembled_df = assemble_vectors(df, ['x1'], 'y')
#Build regression model
from pyspark.ml.regression import LinearRegression
reg = LinearRegression(featuresCol='features', labelCol='y')
reg_model = reg.fit(assembled_df) # fit model
# print coefficient and intercept
print(reg_model.coefficients[0], reg_model.intercept)
# Output: -0.8705560619872369 61.87237921604372
```

```
#prediction result
pred_result = reg_model.transform(assembled_df)
# model summary
reg_summary = reg_model.summary
```

Model Equation

y-hat = -0.8706* x1 + 61.8723

Error

Error is defined as the difference between prediction and actual target. We use the following equations to predict *y-hat* values:

```
Obs1 = -0.8706 * 58.0 + 61.8723 = 11.38
Obs2 = -0.8706 * 37.0 + 61.8723 = 29.66
Obs3 = -0.8706 * 29.0 + 61.8723 = 36.63
Obs4 = -0.8706 * 19.0 + 61.8723 = 45.33
```

We have the predictions now. Let's calculate the error for each prediction.

```
Error1 = 12 - 11.38 = 0.62
Error2 = 27 - 29.66 = -2.66
Error3 = 39 - 36.62 = 2.37
Error4 = 45 - 45.33 = -0.33
```

What happens when we add errors to quantify our model prediction results?

```
Total error= 0.62 - 2.66 + 2.37 - 0.33 = 0.0078
```

This does not make sense because the error does not properly quantify our evaluation results. Positive errors cancel out negative ones, making our error value very small. *This effect is more profound when you have outliers in your data, where one or two values might compensate for the rest of the errors*. Therefore, we need to use some other metric to quantify our errors. Table 6-3 summarizes each selection metric and shows how to use them to evaluate continuous model performance.

Table 6-3. *Selection Metrics*

Metric	Model Performance
Mean squared error (MSE)	Lower is better
Root mean square error (RMSE)	Lower is better
Mean absolute error	Lower is better
R-squared	Higher is better
Adjusted R-squared	Higher is better
Explained variance	Higher is better

Mean Squared Error (MSE)

Mean squared error takes the average of the squared errors. First, let's calculate the squared error. We will use the errors calculated earlier.

```
Sum Squared Error (SSE) = (0.62)² + (-2.66)² + (2.37)² + (-0.33)² = 13.21
```

The mean squared error is the average of the above value.

```
Mean Squared Error (MSE) = 13.21/4 = 3.303
```

Let's do the same calculation using PySpark.

```
from pyspark.ml.evaluation import RegressionEvaluator
evaluator = RegressionEvaluator(labelCol='y', predictionCol='prediction',
metricName='mse')
evaluator.evaluate(pred_result)
# using model summary
print('Mean Squared error', reg_summary.meanSquaredError)
```

Root Mean Squared Error (RMSE)

Root mean squared error takes the square root of the mean squared error (MSE).

$$\text{RMSE} = \sqrt[2]{MSE} = \sqrt[2]{3.303} = 1.817$$

Let's do the same calculation using PySpark.

```
evaluator = RegressionEvaluator(labelCol='y', predictionCol='prediction',
metricName='rmse')
evaluator.evaluate(pred_result)
# using model summary
print('Root mean squared error', reg_summary.rootMeanSquaredError)
```

Mean Absolute Error (MAE)

Mean absolute error takes the average of the absolute value of the errors. Let's calculate the absolute error for these predictions.

```
Sum Absolute Error (SAE) = 0.62 + 2.66 + 2.37 + 0.33 = 5.989
```

Mean absolute error takes the average of this value.

```
Mean absolute error = 5.989/4 = 1.497
```

Let's do the same calculation in PySpark.

```
evaluator = RegressionEvaluator(labelCol='y', predictionCol='prediction',
metricName='mae')
evaluator.evaluate(pred_result)
# using model summary
print('Mean Absolute error', reg_summary.meanAbsoluteError)
```

R-squared (R^2)

R-squared is the coefficient of the determination of a regression model. It measures the goodness of fit; that is, how well the model fits the data. There are various ways to represent the R-squared equation.

Using Model Fit

Say,

```
SST = SSM + SSE
where
SST = Sum of squares total
```

```
SSM = Sum of squares model
SSE = Sum of squares error
Then
R-squared = SSM/ SST = 1 – (SSE/SST)
```

Using Model Variance

```
var(model) = variance of model = MSE
var(mean) = Average of sum of squared error around target mean (MSM)
R squared = 1 – (var(model)/var(mean)) = 1 – (MSE/MSM)
```

Let's calculate it manually. This time, we use the model variance method since we already calculated MSE.

```
Mean of target (mean) = (12 + 27 + 39 + 45)/4 = 30.75
SS(mean) = (12 – 30.75)² + (27 – 30.75)² + (39 – 30.75)² + (45 – 30.75)²
var(mean) = (351.56 + 14.06 + 68.06 + 203.06)/4 = 636.75/4 = 159.19
var(model) = MSE = 3.303
R-squared = 1 – (3.303/159.19) = 0.9792
```

Note SS (mean) is the same as SST. Let's do the same calculation in PySpark.

```
evaluator = RegressionEvaluator(labelCol='y', predictionCol='prediction',
metricName='r2')
evaluator.evaluate(pred_result)
print('R squared', reg_summary.r2)
```

Explained Variance (Var)

Explained variance is the difference between the baseline model variance and the fitted model variance.

```
Explained variance (var) = var(mean) – var(model)
Var = 159.19 – 3.303 = 155.884
```

Let's do the same calculation in PySpark.

```
# option available for Pyspark 3.0 and above
evaluator = RegressionEvaluator(labelCol='y', predictionCol='prediction',
metricName='var')
evaluator.evaluate(pred_result)
# PySpark < 3.0
print('Explained Variance', reg_summary.explainedVariance)
```

Adjusted R-Squared (Adj. R^2)

R-squared is a biased estimate since the number of parameters influences the value. When you add a noisy parameter or a completely random parameter, the R-squared value increases. Therefore, it is good to adjust the R-squared value based on the number of parameters in the model. The formula is provided here:

$$R^2_{adjusted} = 1 - \frac{\left(1 - R^2\right)\left(N - 1\right)}{N - p - 1}$$

N – Number of observations (4 in this example)
p – Number of parameters (1 in this example, 'x1')
$R^2_{adjusted}$ = 1 - (((1-0.97924) *3)/ (4-1-1)) = 0.96886
Let's do the calculation using PySpark.

```
print('R squared', reg_summary.r2adj)
```

Binary Target

When it comes to having a binary target, PySpark offers `BinaryClassificationEvaluator` and `MulticlassClassificationEvaluator` to evaluate model performance. These options are available in the PySpark *ML Tuning* package. In addition, you can use the `BinaryClassificationMetrics` and `MulticlassMetrics` options. Unfortunately, PySpark does not support all the evaluation methods used in the Scala version. Sometimes, you need to use workarounds to get the metrics. We suggest you convert the predictions to a pandas dataset and use `scikit-learn` to generate the metrics. This is the alternative approach. In this book, we use the native PySpark approach. Table 6-4 provides the sample data (OR Logic Gate).

Table 6-4. *Sample Data*

x1	x2	Target
0	0	0
0	1	1
1	0	1
1	1	1

Figure 6-12 shows the data using the x1 and x2 features plotted against the target.

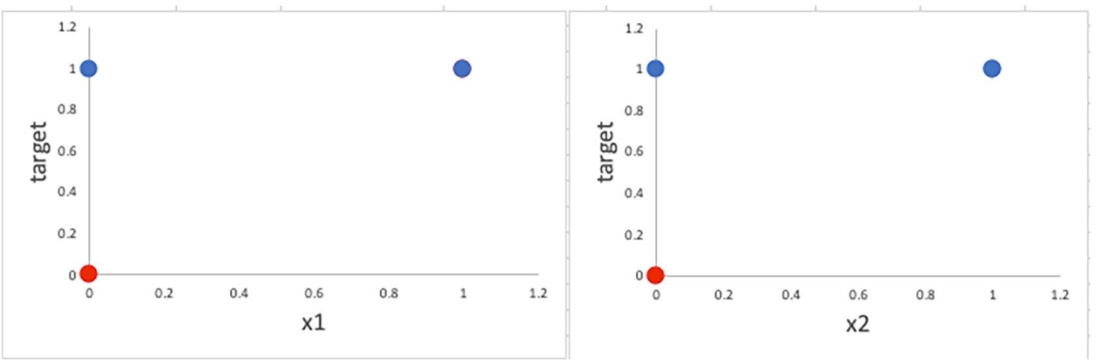


Figure 6-12. *Plot of sample data*

We use x1 vs target to evaluate the classification metrics.

Code to Replicate the Output Using PySpark

```
from pyspark.sql.types import IntegerType, DoubleType
from pyspark.sql.types import StructField, StructType
cSchema = StructType([StructField("x1", IntegerType())\
                      ,StructField("x2", IntegerType())\
                      ,StructField("target", IntegerType())])
df_list = [[0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 1]]
df = spark.createDataFrame(df_list, schema=cSchema)
assembled_df = assemble_vectors(df, ['x1'], 'target')
from pyspark.ml.classification import LogisticRegression
clf = LogisticRegression(featuresCol='features', labelCol='target')
```

```
clf_model = clf.fit(assembled_df)
# Print the coefficient and intercept
print(clf_model.coefficients[0], clf_model.intercept)
#Output: 14.49962091530358, -7.57e-09
pred_result = clf_model.transform(assembled_df)
```

Model Equation

y-hat = 14.49* x1 - 7.57e-09

y-hat Prediction to Probability to Final Prediction

We use the Sigmoid function to convert the `y-hat` predictions to probabilities.

$$Sigmoid\ function, Probability = \left(\frac{1}{1+\exp(-x)} \right)$$

Now we can calculate predictions using the preceding y-hat formula.

```
Obs1 = 22.45* 0 + 0.0000021 = 0.0000021
```

Similarly, we get 0.0000021, 22.45, and 22.45 for `Obs2`, `Obs3`, and `Obs4`, respectively. Now, we convert the predictions to probabilities using the Sigmoid function. This results in values of 0.5, 0.5, 1.0, and 1.0, respectively. We convert the probabilities to predictions by using a cutoff value. Most of the algorithms, by default, use 0.5 as the cut-off. This cut-off can be modified to suit your case. We use 0.5 in this example as well.

```
probability >= cut-off = 1
probability < cut-off = 0
```

Based on this cut-off, we get the *final prediction* as 1, 1, 1, and 1, respectively.

Confusion Matrix

We then overlay our predictions against the actual target values to get the confusion matrix shown in Figure 6-13. The code that generates the matrix is provided for reference.

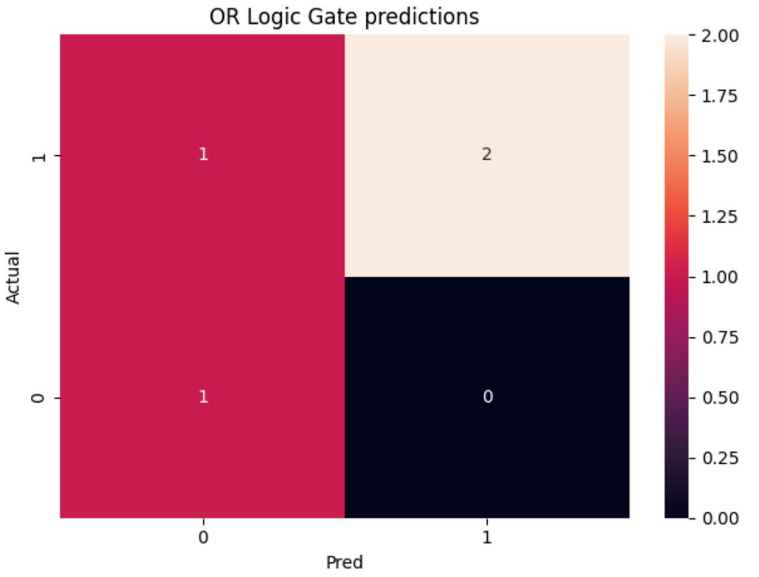


Figure 6-13. *Confusion matrix*

```
from pyspark.mllib.evaluation import BinaryClassificationMetrics,
MulticlassMetrics
# select the required columns from prediction result
predictionAndTarget = pred_result.select("prediction", "target")
predictionAndTarget = predictionAndTarget.withColumn("target",
predictionAndTarget["target"].cast(DoubleType()))
predictionAndTarget = predictionAndTarget.withColumn("prediction",
predictionAndTarget["prediction"].cast(DoubleType()))
#convert to RDD for MulticlassMetrics
metrics = MulticlassMetrics(predictionAndTarget.rdd.map(tuple))
# confusion matrix
cm = metrics.confusionMatrix().toArray()
#plotting functions
import seaborn as sns
import matplotlib.pyplot as plt
def make_confusion_matrix_chart(cf_matrix):
    list_values = ['0', '1']
    plt.subplot(111)
    sns.heatmap(cf_matrix, annot=True, yticklabels=list_values,
                               xticklabels=list_values, fmt='g')
```

```
    plt.ylabel("Actual")
    plt.xlabel("Pred")
    plt.ylim([0,2])
    plt.title('OR Logic Gate predictions')
    plt.tight_layout()
    return None
make_confusion_matrix_chart(cm) #make confusion matrix plot
```

A confusion matrix in abstract terms looks like the following.

		Predicted	
		Negative	**Positive**
Actual	Positive	False Negative (FN)	True Positive (TP)
	Negative	True Negative (TN)	False Positive (FP)

By overlaying the abstract term matrix on this result matrix, we get the following:

```
True Positive, TP = 2
True Negative, TN = 1
False Positive, FP = 0
False Negative, FN = 1
```

Now we calculate the classification metrics one by one to evaluate this model (see Table 6-5). Before that, we go through the code to calculate them all at once.

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator
acc = metrics.accuracy
misclassification_rate = 1 - acc
precision = metrics.precision(1.0)
recall = metrics.recall(1.0)
f1 = metrics.fMeasure(1.0)
evaluator = BinaryClassificationEvaluator(labelCol='target',
rawPredictionCol='rawPrediction', metricName='areaUnderROC')
roc = evaluator.evaluate(pred_result)
print(acc, misclassification_rate, precision, recall, f1, roc)
```

Table 6-5. *Classification Metrics*

Metric	Range	Model Performance	When To Use It	Examples
Accuracy	0 to 1	Higher is better.	All classes are important.	Target distribution is somewhat balanced.
Misclassification rate	0 to 1	Lower is better.	All classes are important.	Target distribution is somewhat balanced.
Precision	0 to 1	Higher is better.	Cost of false positives is high.	Scientific tests (pregnancy, COVID-19).
Recall	0 to 1	Higher is better.	Cost of false negative is high.	Imbalanced data (fraud detection).
F1-score	0 to 1	Higher is better.	False positives and false negatives costs are both high.	Information retrieval task.
ROC or AUC	0.5 to 1	Higher is better.	All classes are important.	Presentations, business stakeholders.
KS Statistic	-	Higher is better.	Separation between 0 and 1 is important.	Two sample analysis.
Deciles	-	Ranking of probabilities from highest to lowest into ten segments. Ranking should not break.	Ranking of observations matters.	Marketing campaigns.

(*continued*)

Table 6-5. (*continued*)

Metric	Range	Model Performance	When To Use It	Examples
Confusion matrix	-	Results (TP, TN, FP, FN)	All classes are important.	Presentations, business stakeholders.
ROC curve	0.5 to 1	Performance graph.	All classes are important.	Presentations, business stakeholders.
Precision recall curve	Target distribution of positive classes (for balanced classes 0.5) to 1	Performance graph.	Imbalanced classes.	Information retrieval task.
Gini coefficient (out of scope of this book)	0 to 1 Formula: (2 * AUC – 1)	Higher is better.	All classes are important.	Credit score models.

Accuracy

Accuracy measures how many times the algorithm prediction matches the actual target. It is defined by the following formula:

$$Accuracy = \left(\frac{TP + TN}{TP + TN + FP + FN} \right)$$

The calculated accuracy value for this model is `0.75`. Alternatively, you can think that one out of four predictions is wrong. Therefore, the accuracy is `0.75`.

Misclassification Rate

The misclassification rate measures *how many times the algorithm's prediction did not match the actual target*. It is also known as the error rate. It is defined by the following formula:

```
Misclassification rate = 1 – Accuracy
```

The misclassification rate for the model is `0.25`. Alternatively, you can think that one out of four predictions is wrong. Therefore, the error rate is `0.25`.

Precision

Precision measures *how many times the algorithm returns a relevant match based on its predictions*. It is also known as the positive predictive value. It is defined by the following formula:

$$Precision = \left(\frac{TP}{TP + FP} \right)$$

The model precision here is `0.75`.

Recall

Recall measures *how many times the algorithm returns a relevant match based on all the relevant items*. It is also known as sensitivity or the true positive rate. It is defined by the following formula:

$$Recall = \left(\frac{TP}{TP + FN} \right)$$

The model recall is `1`.

F1-score

F1-score measures the harmonic mean between precision and recall. It is defined by the following formula:

$$F1 - score = 2 * \left(\frac{Precision * Recall}{Precision + Recall} \right)$$

The model f1-score is `0.86`.

Receiver Operating Characteristics (ROC)/Area Under the Curve (AUC)

ROC or AUC measures the area under the sensitivity and 1-specificity curve. It is shown with a light shade in Figure 6-14.

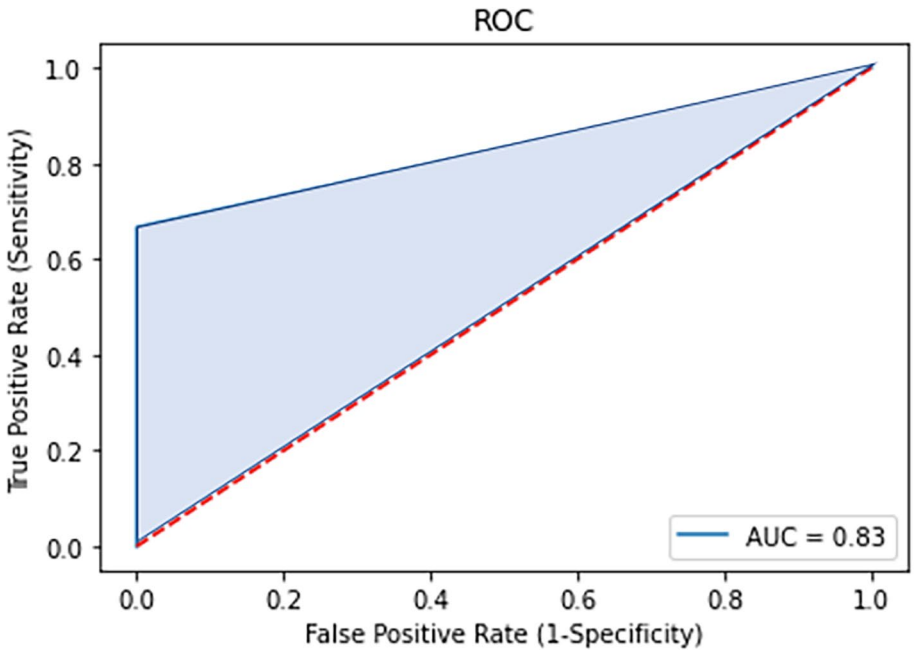


Figure 6-14. *ROC*

How is this chart generated? Well, you already know about the sensitivity (recall). Specificity is defined by the following formula:

$$Specificity = \left(\frac{TN}{TN + FP} \right) = 1 - FPR$$

where

$$False\ Positive\ Rate = \left(\frac{FP}{FP + TN} \right)$$

Remember the cut-off we used to convert probability to prediction; we set it to 0.5 for the initial calculation. As mentioned, this value is customizable. Using a range of values from `0 to 1 (0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1)` as cut-off values, you can calculate the sensitivity and the specificity using the formula.

Then, you create a graph as shown using the calculated values of sensitivity and specificity for all the ranges. For finer details in the graph, you can increase the range of values in between 0 and 1. Finally, you calculate the area under the curve formed by the graph.

The red line represents a *random model*. Since the plot is a square, the area of the triangle formed by the red line equals `0.5 * base * height = 0.5 * 1 * 1 = 0.5`. This means that the baseline value of ROC is 0.5. The area of the square is 1. Therefore, the maximum value of ROC is 1. To calculate the area formed by the curve (blue line) and the red line by hand, you can create squares and triangles and calculate the area of each figure separately. By adding all the areas, you'll get the final ROC value.

In PySpark, you can execute the following code to draw the ROC plot:

```
from pyspark.mllib.evaluation import BinaryClassificationMetrics
# calculate fpr, tpr using CurveMetrics
class CurveMetrics(BinaryClassificationMetrics):
    def __init__(self, *args):
        super(CurveMetrics, self).__init__(*args)
# Convert RDD results to a list of points
    def _to_list(self, rdd):
        points = []
        results_collect = rdd.collect()
        for row in results_collect:
            points += [(float(row._1()), float(row._2()))]
        return points
# Retrieve curve points
    def get_curve(self, method):
        rdd = getattr(self._java_model, method)().toJavaRDD()
        return self._to_list(rdd)
# use the probability colum to apply cut-offs
preds = pred_result.select('target','probability').rdd.map(lambda row:
(float(row['probability'][1]), float(row['target'])))
# Returns as a list (false positive rate, true positive rate)
points = CurveMetrics(preds).get_curve('roc')
# make plot
import matplotlib.pyplot as plt
plt.figure()
x_val = [x[0] for x in points]
y_val = [x[1] for x in points]
plt.title('ROC')
plt.xlabel('False Positive Rate (1-Specificity)')
```

```
plt.ylabel('True Positive Rate (Sensitivity)')
plt.plot(x_val, y_val, label = 'AUC = %0.2f' % roc)
plt.plot([0, 1], [0, 1], color='red', linestyle='--')
plt.legend(loc = 'lower right')
```

What Happens to the ROC Curve When the Classifier Is Bad?

The curve will overlap the red line random model, and thus, the value of ROC will be close to 0.5.

Precision Recall Curve

The precision recall curve is the trade-off curve between precision and recall. It is useful for imbalanced classification exercises. Similar to ROC, precision and recall values are calculated for different cut-offs and a plot is generated, as shown in Figure 6-15. The red line is generated based on the target distribution of ones in the data. In this case, 50 percent of the target is 1. Therefore, the model baseline is set to 0.5.

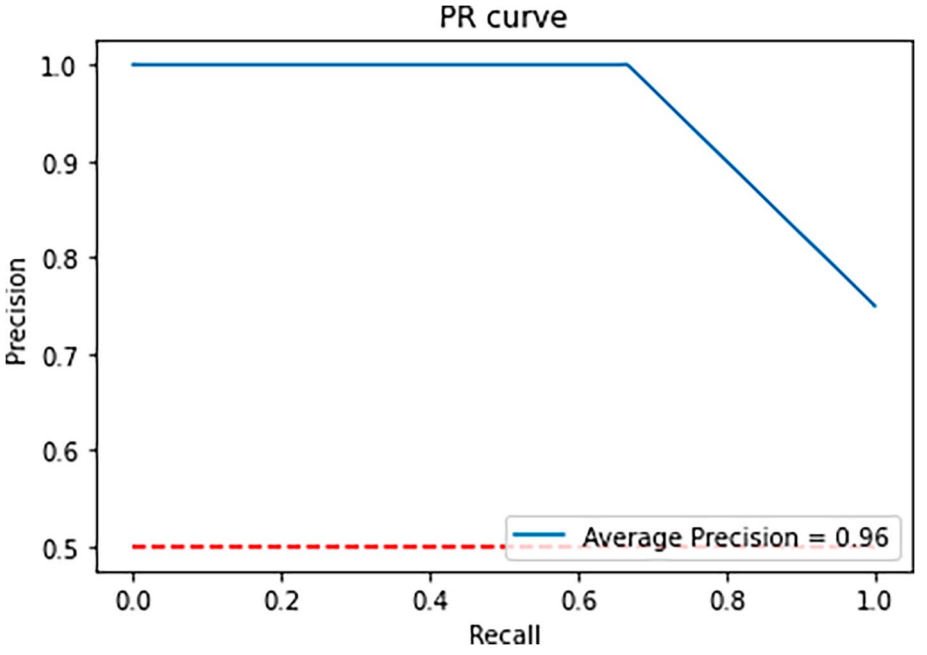


Figure 6-15. *Precision recall curve*

You can calculate the area under the blue line by averaging the precision values at different cut-offs. For this chart, it is easy to calculate the area by hand. The area of the left-out triangle on top of the figure is `0.33 * 0.25 * 0.5 = 0.0416`. Therefore, the area under the PR curve is `1 - 0.0416 = 0.9584`. We can use PySpark to accomplish the same task:

```
# Evaluate using BinaryClassificationEvaluator for PR curve
evaluator = BinaryClassificationEvaluator(labelCol='target', rawPrediction
Col='rawPrediction', metricName='areaUnderPR')
pr = evaluator.evaluate(pred_result)
preds = pred_result.select('target','probability').rdd.map(lambda row:
(float(row['probability'][1]), float(row['target'])))
# Calculate points for PR curve using custom CurveMetrics class
points = CurveMetrics(preds).get_curve('pr')
plt.figure()
x_val = [x[0] for x in points]
y_val = [x[1] for x in points]
plt.title('PR curve')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.plot(x_val, y_val, label = 'Average Precision = %0.2f' % pr)
plt.plot([0, 1], [0.5, 0.5], color='red', linestyle='--')
plt.legend(loc = 'lower right')
```

What Happens to the PR Curve When the Classifier Is Bad?

The area under the PR curve will resemble the area of a distribution with all target values equal to 1. In this case, the value will be close to 0.5. We can prove that using a random value generator (see Figure 6-16).

```
from sklearn.metrics import precision_recall_curve, average_precision_score
# random target and probabilities
rand_y = [random.choice([1, 0]) for i in range(0, 100)]
rand_prob = [random.uniform(0, 1) for i in range(0, 100)]
# Calculate precision, recall, and average precision score
rand_precision, rand_recall, _ = precision_recall_curve(rand_y, rand_prob)
pr = average_precision_score(rand_y, rand_prob)
```

```
#plot random predictions
plt.figure()
plt.title('PR curve')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.plot(rand_recall, rand_precision, label = 'Average Precision =
%0.2f' % pr)
plt.plot([0, 1], [0.5, 0.5], color='red', linestyle='--')
plt.legend(loc = 'lower right')
```

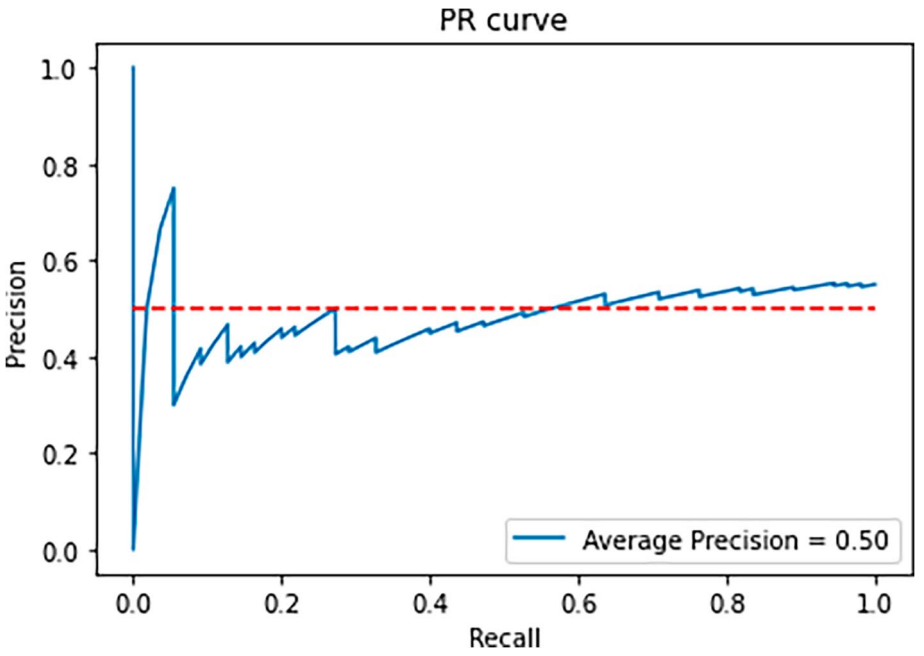


Figure 6-16. *PR curve with bad classifier*

Kolmogorov Smirnov (KS) Statistic and Deciles

The KS statistic measures the difference in the distribution of target versus non-target values. It is a non-parametric evaluation. To perform KS, you need to first create deciles.

Deciles

To get the deciles for a model, follow these steps:

1) Apply the model on the dataset.
2) Sort the scores based on descending probability (ties handled implicitly).
3) Divide the scores into ten buckets (deciles).

Now you have the deciles. Deciles assist you with a variety of analyses, especially in campaigns. Based on a decile, you can measure the conversion ratio in the top 10 percent, top 30 percent, and so on. In addition, you can measure lift and actual versus predicted stats.

KS Statistics

We compute the cumulative distribution of target and non-target values by decile. The difference between the two values gives you the spread of the distribution, and the maximum value of spread is the KS.

PySpark does not provide a straightforward method to perform this analysis. We have provided a workaround. For the following work, we used the bank dataset.

```
#load dataset, cleanup and fit model
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler
from pyspark.ml import Pipeline
from pyspark.sql.functions import countDistinct
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.sql import functions as F
import numpy as np
filename = "bank-full.csv"
spark = SparkSession.builder.getOrCreate()
df = spark.read.csv(filename, header=True, inferSchema=True, sep=';')
df = df.withColumn('label', F.when(F.col("y") == 'yes', 1).otherwise(0))
df = df.drop('y')
```

```
#assemble individual columns to one column - 'features'
def assemble_vectors(df, features_list, target_variable_name):
    stages = []
    #assemble vectors
    assembler = VectorAssembler(inputCols=features_list,
    outputCol='features')
    stages = [assembler]
    #select all the columns + target + newly created 'features' column
    selectedCols = [target_variable_name, 'features']
    #use pipeline to process sequentially
    pipeline = Pipeline(stages=stages)
    #assembler model
    assembleModel = pipeline.fit(df)
    #apply assembler model on data
    df = assembleModel.transform(df).select(selectedCols)
    return df
# Select relevant features and assemble into vectors
df = df.select(['education', 'age', 'balance', 'day', 'duration',
'campaign', 'pdays', 'previous', 'label'])
features_list = ['age', 'balance', 'day', 'duration', 'campaign', 'pdays',
'previous']
assembled_df = assemble_vectors(df, features_list, 'label')
clf = LogisticRegression(featuresCol='features', labelCol='label')
train, test = assembled_df.randomSplit([0.7, 0.3], seed=12345)
clf_model = clf.fit(train)
#Deciling and KS calculation begins here
from pyspark.sql import Window
def create_deciles(df, clf, score, prediction, target, buckets):
    # get predictions from model
    pred = clf.transform(df)
    #probability of 1's, prediction and target
    pred = pred.select(F.col(score), F.col(prediction), F.col(target)).
    rdd.map(lambda row: (float(row[score][1]), float(row['prediction']),
    float(row[target])))
    predDF = pred.toDF(schema=[score, prediction, target])
```

```
# remove ties in scores work around
window = Window.orderBy(F.desc(score))
predDF = predDF.withColumn("row_number", F.row_number().over(window))
predDF.cache()
predDF = predDF.withColumn("row_number", predDF['row_number'].
cast("double"))
# partition into 10 buckets
window2 = Window.orderBy("row_number")
final_predDF = predDF.withColumn("deciles", F.ntile(buckets).
over(window2))
final_predDF = final_predDF.withColumn("deciles", final_
predDF['deciles'].cast("int"))
# create non target column
final_predDF = final_predDF.withColumn("non_target", 1 - final_
predDF[target])
final_predDF.cache()    #final predicted df
#ks calculation starts here
temp_deciles = final_predDF.groupby('deciles').agg(F.sum(target).
alias(target)).toPandas()
non_target_cnt = final_predDF.groupby('deciles').agg(F.sum
('non_target').alias('non_target')).toPandas()
temp_deciles = temp_deciles.merge(non_target_cnt, on='deciles',
how='inner')
temp_deciles = temp_deciles.sort_values(by='deciles', ascending=True)
temp_deciles['total'] = temp_deciles[target] + temp_deciles
['non_target']
temp_deciles['target_%'] = (temp_deciles[target] / temp_
deciles['total'])*100
temp_deciles['cum_target'] = temp_deciles[target].cumsum()
temp_deciles['cum_non_target'] = temp_deciles['non_target'].cumsum()
temp_deciles['target_dist'] = (temp_deciles['cum_target']/
temp_deciles[target].sum())*100
temp_deciles['non_target_dist'] = (temp_deciles['cum_non_target']
/temp_deciles['non_target'].sum())*100
temp_deciles['spread'] = temp_deciles['target_dist'] -
temp_deciles['non_target_dist']
```

```
    decile_table=temp_deciles.round(2)
    decile_table = decile_table[['deciles', 'total', 'label', 'non_target',
    'target_%', 'cum_target', 'cum_non_target', 'target_dist', 'non_target_
    dist', 'spread']]
    print("KS Value - ", round(temp_deciles['spread'].max(), 2))
    return final_predDF, decile_table
#create deciles on the train and test datasets
pred_train, train_deciles = create_deciles(train, clf_model, 'probability',
'prediction', 'label', 10)
pred_test, test_deciles = create_deciles(test, clf_model, 'probability',
'prediction', 'label', 10)
#pandas styling functions
from collections import OrderedDict
import pandas as pd
import sys
%matplotlib inline
def plot_pandas_style(styler):
    from IPython.core.display import HTML

html = '\n'.join([line.lstrip() for line in styler.render().split('\n')])
    return HTML(html)
def highlight_max(s,color='yellow'):
    '''
    highlight the maximum in a Series yellow.
    '''
    is_max = s == s.max()

return ['background-color: {}'.format(color) if v else '' for v in is_max]
def decile_labels(agg1, target, color='skyblue'):
    agg1 = agg1.round(2)

agg1 = agg1.style.apply(highlight_max, color = 'yellow',
subset=['spread']).set_precision(2)
    agg1.bar(subset=[target], color='{}'.format(color), vmin=0)
    agg1.bar(subset=['total'], color='{}'.format(color), vmin=0)
    agg1.bar(subset=['target_%'], color='{}'.format(color), vmin=0)
    return agg1
```

```
#train deciles and KS
plot_decile_train = decile_labels(train_deciles, 'label', color='skyblue')
plot_pandas_style(plot_decile_train)
```

	deciles	total	label	non_target	target_%	cum_target	cum_non_target	target_dist	non_target_dist	spread
0	1	3156.00	1494.00	1662.00	47.34	1494.00	1662.00	39.54	5.98	33.56
1	2	3156.00	806.00	2350.00	25.54	2300.00	4012.00	60.88	14.44	46.44
2	3	3156.00	491.00	2665.00	15.56	2791.00	6677.00	73.88	24.03	49.84
3	4	3156.00	352.00	2804.00	11.15	3143.00	9481.00	83.19	34.13	49.06
4	5	3156.00	250.00	2906.00	7.92	3393.00	12387.00	89.81	44.59	45.22
5	6	3156.00	170.00	2986.00	5.39	3563.00	15373.00	94.31	55.34	38.97
6	7	3156.00	98.00	3058.00	3.11	3661.00	18431.00	96.90	66.34	30.56
7	8	3156.00	56.00	3100.00	1.77	3717.00	21531.00	98.39	77.50	20.88
8	9	3156.00	42.00	3114.00	1.33	3759.00	24645.00	99.50	88.71	10.79
9	10	3155.00	19.00	3136.00	0.60	3778.00	27781.00	100.00	100.00	0.00

Figure 6-17. *Training dataset*

```
#test deciles and KS
plot_decile_test = decile_labels(test_deciles, 'label', color='skyblue')
plot_pandas_style(plot_decile_test)
```

	deciles	total	label	non_target	target_%	cum_target	cum_non_target	target_dist	non_target_dist	spread
0	1	1366.00	612.00	754.00	44.80	612.00	754.00	40.50	6.21	34.29
1	2	1366.00	292.00	1074.00	21.38	904.00	1828.00	59.83	15.06	44.77
2	3	1365.00	213.00	1152.00	15.60	1117.00	2980.00	73.92	24.54	49.38
3	4	1365.00	150.00	1215.00	10.99	1267.00	4195.00	83.85	34.55	49.30
4	5	1365.00	104.00	1261.00	7.62	1371.00	5456.00	90.73	44.94	45.80
5	6	1365.00	59.00	1306.00	4.32	1430.00	6762.00	94.64	55.70	38.94
6	7	1365.00	38.00	1327.00	2.78	1468.00	8089.00	97.15	66.63	30.53
7	8	1365.00	24.00	1341.00	1.76	1492.00	9430.00	98.74	77.67	21.07
8	9	1365.00	16.00	1349.00	1.17	1508.00	10779.00	99.80	88.78	11.02
9	10	1365.00	3.00	1362.00	0.22	1511.00	12141.00	100.00	100.00	0.00

Figure 6-18. *Test dataset*

The KS values for training and testing are 50.28 and 49.63, respectively, as shown in Figures 6-17 and 6-18. The testing KS value lies within ± 3 values from the training KS value. In addition, you can also see that the deciles do not have breaks in ranking. Therefore, this model is stable.

Actual vs Predicted, Gains Chart, Lift Chart

Let's implement these metrics in PySpark and then look at each one in detail. We use the output from the previous step.

```
import matplotlib.pyplot as plt
def plots(agg1,target,type):
    plt.figure(1,figsize=(20, 5))
    plt.subplot(131)
    plt.plot(agg1['DECILE'],agg1['ACTUAL'],label='Actual')
    plt.plot(agg1['DECILE'],agg1['PRED'],label='Pred')
    plt.xticks(range(10,110,10))
    plt.legend(fontsize=15)
    plt.grid(True)
    plt.title('Actual vs Predicted', fontsize=20)
    plt.xlabel("Population %",fontsize=15)
    plt.ylabel(str(target) + " " + str(type) + " %",fontsize=15)
    plt.subplot(132)
    X = agg1['DECILE'].tolist()
    X.append(0)
    Y = agg1['DIST_TAR'].tolist()
    Y.append(0)
    plt.plot(sorted(X),sorted(Y))
    plt.plot([0, 100], [0, 100],'r--')
    plt.xticks(range(0,110,10))
    plt.yticks(range(0,110,10))
    plt.grid(True)
    plt.title('Gains Chart', fontsize=20)
    plt.xlabel("Population %",fontsize=15)
    plt.ylabel(str(target) + str(" DISTRIBUTION") + " %",fontsize=15)

plt.annotate(round(agg1[agg1['DECILE'] == 30].DIST_TAR.
item(),2),xy=[30,30],

xytext=(25, agg1[agg1['DECILE'] == 30].DIST_TAR.item() + 5),
fontsize = 13)

plt.annotate(round(agg1[agg1['DECILE'] == 50].DIST_TAR.item(),2),
xy=[50,50],

xytext=(45, agg1[agg1['DECILE'] == 50].DIST_TAR.item() + 5),
fontsize = 13)
```

```
    plt.subplot(133)
    plt.plot(agg1['DECILE'],agg1['LIFT'])
    plt.xticks(range(10,110,10))
    plt.grid(True)
    plt.title('Lift Chart', fontsize=20)
    plt.xlabel("Population %",fontsize=15)
    plt.ylabel("Lift",fontsize=15)
    plt.tight_layout()
# aggregations for actual vs predicted, gains and lift
def gains(data, decile_df, decile_by, target, score):
    agg1 = pd.DataFrame({},index=[])
    agg1 = data.groupby(decile_by).agg(F.avg(target).alias('ACTUAL')).
    toPandas()
    score_agg = data.groupby(decile_by).agg(F.avg(score).alias('PRED')).
    toPandas()
    agg1 = agg1.merge(score_agg, on=decile_by, how='inner').merge
(decile_df, on=decile_by, how='inner')
    agg1 = agg1.sort_values(by=decile_by, ascending=True)
    agg1 = agg1[[decile_by, 'ACTUAL', 'PRED', 'target_dist']]
    agg1 = agg1.rename(columns={'target_dist':'DIST_TAR', 'deciles':
    'DECILE'})
    decile_by = 'DECILE'
    agg1[decile_by] = agg1[decile_by]*10
    agg1['LIFT'] = agg1['DIST_TAR']/agg1[decile_by]
    agg1.columns = [x.upper() for x in agg1.columns]
    plots(agg1,target,'Distribution')
# train metrics
gains(pred_train, train_deciles, 'deciles', 'label','probability')
```

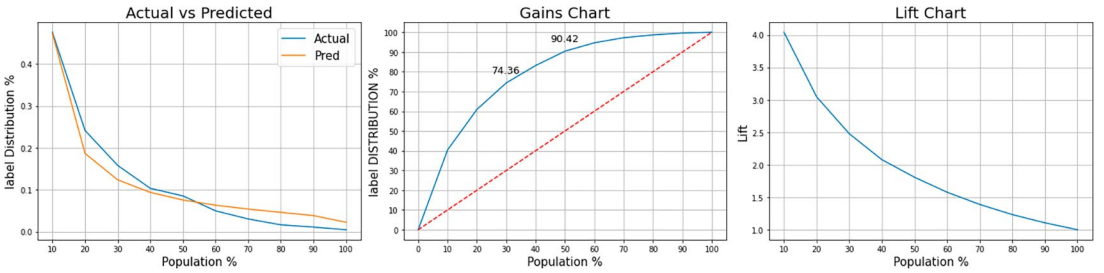


Figure 6-19a. *Actual versus predicted, gains chart, lift chart*

```
#test metrics
gains(pred_test, test_deciles, 'deciles', 'label','probability')
```

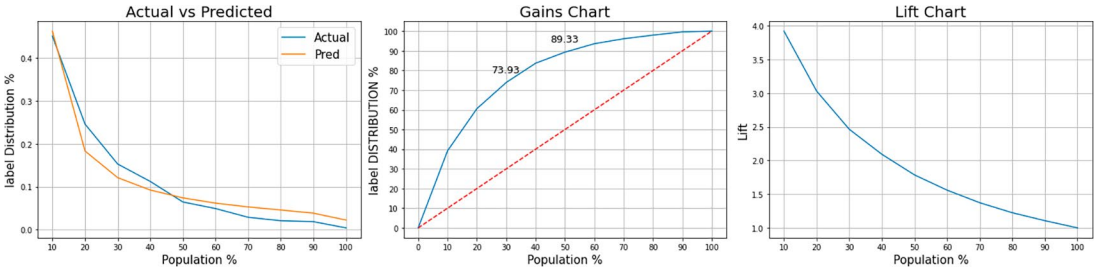


Figure 6-19b. *Actual versus predicted, gains chart, lift chart*

Okay, now you know what these metrics look like. Let's dive into each one of them.

Actual Versus Predicted

This overlays the mean distribution of targets and predictions on top of one another. When the predicted value is very close to the actual value, it signifies that the model is doing a great job, and vice versa.

Gains Chart

This chart specifies the conversion ratio that can be achieved by targeting a certain percentage of the population. In this example here, by reaching out to the top 30 percent of customers, the conversion of targets is 74 percent. This means that almost three-fourths of the targets are in the top three deciles in the model. The higher the gains chart value, the better the model. This is a very useful metric for explaining your model to the marketing team, since they focus more on conversion-based metrics.

Lift Chart

A lift chart is one more way to look at the conversion. The baseline model has a lift of 1. It means that when you target 10 percent of the population, you get 10 percent conversion. Similarly, when you target 20 percent, you get 20 percent, and so forth. However, by building a machine learning model, you can achieve 40 percent conversion in the top 10 percent of the population. Therefore, the lift achieved by your model in the top 10 percent of the population is four times more than the baseline model. As you go further into the deciles, the lift value reduces for your model and eventually becomes 1. Marketing teams can use this metric to determine their target population during campaigns.

Multiclass, Multilabel Evaluation Metrics

These topics are out of scope for this book. We used some multiclass metrics in binary model evaluation. You can apply the insights gained in this chapter to extend the metrics learned in this chapter for multiclass and multilabel classification tasks.

Summary

- You learned about model complexity in detail.
- You learned about different types of model validation techniques.
- You looked at data leakage and how to handle it for better model performance.
- Finally, you went through model evaluation metrics in detail for regression and binary models.

Great job! In the next chapter, you learn about unsupervised techniques, topic modeling, and recommendations. Keep learning and stay tuned.

CHAPTER 7

Unsupervised Learning and Recommendation Algorithms

This chapter explores popular techniques used in the industry to handle unlabeled data. You will discover how to implement them using PySpark. You will also be introduced to the basics of segmentation and recommendation algorithms and the data preparation involved. By the end of this chapter, you will have an appreciation for unsupervised techniques and their use in day-to-day data science activities.

This chapter covers the following topics:

- K-means
- Latent Dirichlet allocation
- Collaborative filtering using alternating least squares

Segmentation

As the word *segmentation* suggests, it is a technique that divides your population (customers or records in your data) into logical groups with a common need. The needs may vary highly across these logical groups but will vary little within the groups. Let's take a step back and consider why you would need this. If you can identify a homogeneous set of customers with common needs, it helps your business create a strategy to serve customers better and keep them engaged. You can create these segments to understand your customers better or for a specific motive, such as messaging or developing strategies for your application or product.

© Ramcharan Kakarla, Sundar Krishnan, Balaji Dhamodharan and Venkata Gunnu 2024
R. Kakarla et al., *Applied Data Science Using PySpark*, https://doi.org/10.1007/979-8-8688-0820-3_7

Say you are an application developer and have data on how your application is used. It would be interesting to identify clusters of users with similar behavior. Based on how your application is used in different geographical locations, times of usage, age groups, or features used, you can create a strategy for improving the app and customizing the interface for different groups of users to improve engagement. Though it sounds easy, identifying the right clusters can be challenging. You wouldn't want too many or too few clusters. You also want to have a sizeable number of customers in each cluster before taking action. You also need these clusters to be stable. If they change frequently, it may pose a challenge to your intended actions.

Depending on the type of business vertical you are working with, many variables will be available for segmentation. The question becomes, how do you choose your variables for this exercise? Before proceeding into any variable selection methods discussed earlier, it is important to identify base and descriptor variables. Base variables are those used in the segmentations to define clusters. These could be application usage, times used, features used, and so on. Descriptors are not used in the segmentations; they profile and define identified clusters. These descriptors could be any internal key performance indicators that you want to quantify and observe the differences between clusters. For example, if you used application usage data for clustering, you would use application return rate, application churn rate, and application revenue as some of the descriptors.

Again, controlling the descriptor variables for a particular objective can yield different results. There are no hard and fast rules on what variables can be used as bases and descriptors. These are objective specific. You can also drive some of these segmentations based on survey data collected online to identify the customers' changing needs. Data can include geographic, demographic, socioeconomic, application usage, and loyalty information. Surveys can sometimes help collect unobservable data such as lifestyles, preferences, and benefits.

Business rules can also be used for segmenting data, but with increasing troves of data, most organizations prefer using the clustering techniques available. One of the challenges with some of these techniques is that you do not know how many clusters exist in the data. The following data is made up for demonstration purposes. It considers two factors that customers review when purchasing cars. This data has two variables: safety and fuel efficiency. Both these variables were rated by 15 respondents on a scale of 1 to 10. This data is presented in Table 7-1.

Table 7-1. *Survey Data for Car Purchase*

Respondent	Safety	Fuel Efficiency
1	10	10
2	9	9
3	9.5	9.5
4	9.75	9.75
5	8.75	8.75
6	10	1
7	9	1.5
8	9.5	2
9	8.75	1.75
10	9.25	1.25
11	1	10
12	1.5	9
13	2	9.5
14	1.75	8.75
15	1.25	9.25

If you plot the data on a scatter plot, you will immediately recognize three clusters, as shown in Figure 7-1. For respondents in Cluster 1, safety is more important than fuel efficiency. Safety and fuel efficiency are equally important for respondents in Cluster 2. For respondents in Cluster 3, fuel efficiency plays a major role in the purchase of vehicles. What do you do with this information? Car dealers can use this information to market the right cars to customers. They can spend more time explaining the safety features to the respondents in Cluster 1 versus talking about fuel efficiency to the respondents in Cluster 3.

You can make these inferences just by plotting the variables on a twodimensional graph. However, you'll often have multiple variables, and you may be unable to make these judgments visually. In the next sections, we explore techniques for using multidimensional data.

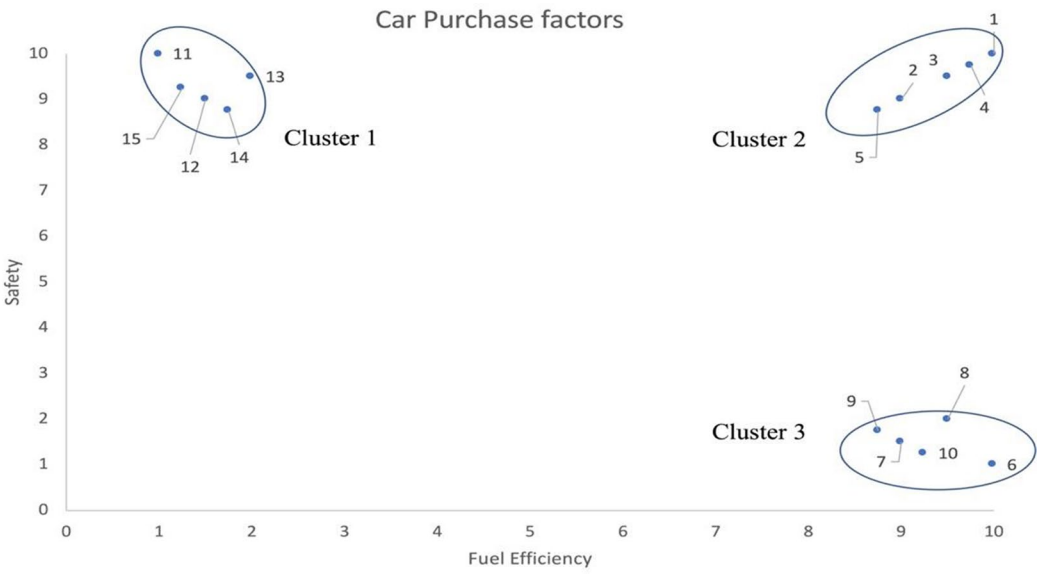


Figure 7-1. *Clusters demonstration*

There is no single clustering solution. In this example, you can broadly classify cars based on size into compacts, sedans, and SUVs. Similarly, you can classify vehicles into luxury/non-luxury and performance/non-performance segments. All of these are, again, valid segments. Similarly, when you cluster data, there may be multiple solutions. The challenging part is to choose the one that works for your business. Descriptors can help you determine the right metrics to help you choose the solution.

Distance Measures

Based on the preceding graph, it's evident that the points close to each other share similar characteristics. In other words, points separated by a small distance are more similar than points separated by larger distances. Distance between points is one of the key concepts used in segmentation to determine similarity. So, how do you calculate the similarity between the points? Multiple techniques are available; one of the most popular is Euclidean distance. Euclidean distance is a geometric distance between two points in a Cartesian coordinate system. The Euclidean distance between two points—a (x_1,y_1) and b (x_2,y_2)—is calculated using the following:

$$\text{Distance}_{ab}=\sqrt{(x_1- x_2)^2 +(y_1- y_2)^2}$$

Let's calculate this distance for Table 7-1 and see how the distances vary across the respondents. Tables 7-2, 7-3, and 7-4 represent the calculated Euclidean distances between all the respondents.

Table 7-2. *Euclidean Distance for Table 7-1 (Partial Output – Part 1)*

	1	2	3	4	5
1	0.00	1.41	0.71	0.35	1.77
2	1.41	0.00	0.71	1.06	0.35
3	0.71	0.71	0.00	0.35	1.06
4	0.35	1.06	0.35	0.00	1.41
5	1.77	0.35	1.06	1.41	0.00
6	9.00	8.06	8.51	8.75	7.85
7	8.56	7.50	8.02	8.28	7.25
8	8.02	7.02	7.50	7.75	6.79
9	8.34	7.25	7.79	8.06	7.00
10	8.78	7.75	8.25	8.51	7.52
11	9.00	8.06	8.51	8.75	7.85
12	8.56	7.50	8.02	8.28	7.25
13	8.02	7.02	7.50	7.75	6.79
14	8.34	7.25	7.79	8.06	7.00
15	8.78	7.75	8.25	8.51	7.52

Table 7-3. *Euclidean Distance for Table 7-1 (Partial Output – Part 2)*

	6	7	8	9	10
1	9.00	8.56	8.02	8.34	8.78
2	8.06	7.50	7.02	7.25	7.75
3	8.51	8.02	7.50	7.79	8.25
4	8.75	8.28	7.75	8.06	8.51
5	7.85	7.25	6.79	7.00	7.52
6	0.00	1.12	1.12	1.46	0.79
7	1.12	0.00	0.71	0.35	0.35
8	1.12	0.71	0.00	0.79	0.79
9	1.46	0.35	0.79	0.00	0.71
10	0.79	0.35	0.79	0.71	0.00
11	12.73	11.67	11.67	11.32	12.03
12	11.67	10.61	10.63	10.25	10.96
13	11.67	10.63	10.61	10.28	10.98
14	11.32	10.25	10.28	9.90	10.61
15	12.03	10.96	10.98	10.61	11.31

Table 7-4. *Euclidean Distance for Table 7-1 (Partial Output – Part 3)*

	11	12	13	14	15
1	9.00	8.56	8.02	8.34	8.78
2	8.06	7.50	7.02	7.25	7.75
3	8.51	8.02	7.50	7.79	8.25
4	8.75	8.28	7.75	8.06	8.51
5	7.85	7.25	6.79	7.00	7.52
6	12.73	11.67	11.67	11.32	12.03
7	11.67	10.61	10.63	10.25	10.96
8	11.67	10.63	10.61	10.28	10.98
9	11.32	10.25	10.28	9.90	10.61
10	12.03	10.96	10.98	10.61	11.31
11	0.00	1.12	1.12	1.46	0.79
12	1.12	0.00	0.71	0.35	0.35
13	1.12	0.71	0.00	0.79	0.79
14	1.46	0.35	0.79	0.00	0.71
15	0.79	0.35	0.79	0.71	0.00

Observing the data in Tables 7-2, 7-3, and 7-4 reasserts the fact that the distance between the similar points is the least. All the short distances are highlighted, and they by no means accidentally form the visual clusters shown in Figure 7-1. Well, you may have a question on how to extend this to multiple variables. You can do that using the extension of Euclidean distance, which is extensible to any number of variables. Say you have three variables. You can calculate the distance between points a (x_1,y_1,z_1) and b (x_2,y_2,z_2) using the following formula:

$$\text{Distance}_{ab}=\sqrt{(x1- x2)^2 +(y1- y2)^2+(z1- z2)^2}$$

Other forms of distance measures exist, such as Manhattan and Chebyshev distances. For interpretation, consider Figure 7-2.

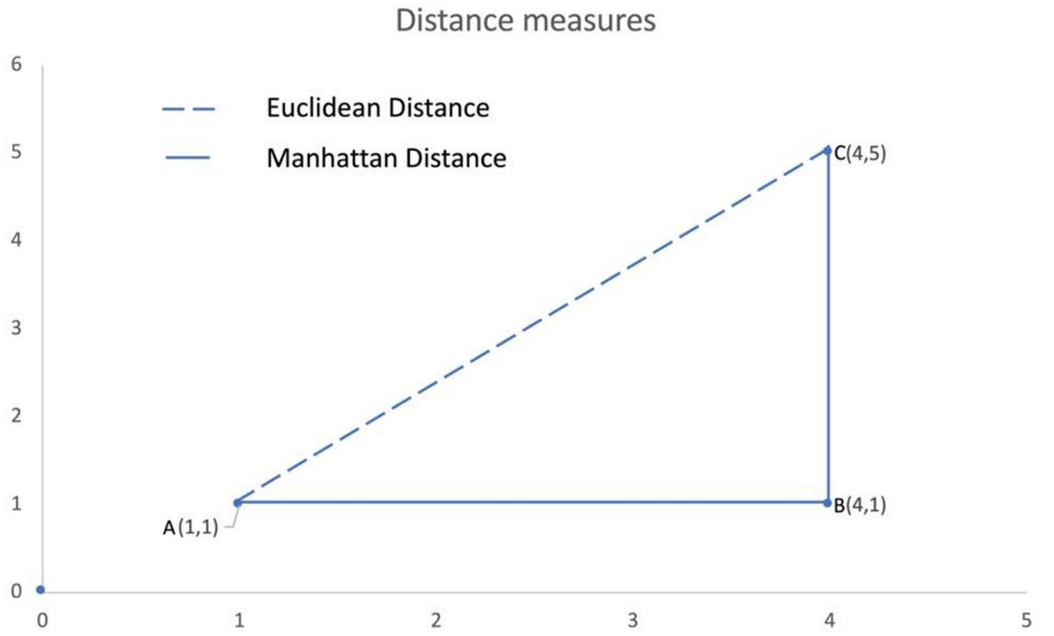


Figure 7-2. *Distance measures*

The Euclidean distance between two points *A* and *C* is 5. You can cross-verify by applying the formula. Manhattan distance between *A* and *C* is calculated as a city block sum of the distance between (*A*,*B*) and (*B*,*C*). It is also called the taxicab metric because of its layout and is calculated for Figure 7-2 as the absolute differences of Cartesian coordinates.

$$\text{Manhattan distance} = |(x2 - x1)| + |(y2-y1)|$$
$$\text{Manhattan distance} = |(4-1)| + |(5-1)|$$
$$\text{Manhattan distance} = 7$$

The Chebyshev distance in a vector space is the distance between two vectors and is the greatest distance of their differences along any coordinate dimension. It can be calculated for Figure 7-2 using the following formula:

$$\text{Chebyshev distance} = \max[(x2 - x1), (y2-y1)]$$
$$\text{Chebyshev distance} = \max[(4 - 1), (5-1)]$$
$$\text{Chebyshev distance} = 4$$

This strategy works great for numerical variables, but how can you apply this technique to categorical variables? Say you have a variable called "previous purchase within 5 years" with values yes/no. How can you use this variable? Remember that we introduced one-hot encoding in an earlier chapter. You can convert this categorical variable yes/no to a numeric variable using one-hot encoding. You will end up with two variables: one for "yes" and another for "no". In practice, you may just need a single variable, as the second variable might be perfectly collinear with the first variable.

If you observe the data carefully, you can see that the earlier variables were on a scale of 1 to 10, and now the new variable "previous purchase within 5 years" has a scale of 0 to 1. Combining these variables will cause a bias, as the new variable will have a negligible effect when you calculate the distances. How do you approach this scenario? Yes, you can use normalization. *Normalization* is a scaling technique that can rescale the existing data in a range of [0,1]. It is calculated using the following formula:

$$X_{normalized} = (X\text{-}X_{min}) / (X_{max} \text{-}X_{min})$$

This is a widely used technique in segmentation. But be cautious, as you might lose information on the outliers.

Another scaling technique that is widely used is *standardization*. Do not confuse standardization with normalization. In standardization, you rescale data to have a mean of 0 and a standard deviation of 1. It is calculated using the following formula:

$$X_{standardized} = X\text{-}\ X_{mean} / X_{standard\_deviation}$$

There might be situations where you only have categorical data. Are there better techniques than one-hot encoding and using Euclidean distance? The answer is yes: Jaccard's coefficient and Hamming distance. We focus here on Jaccard's coefficient. It involves calculating the dissimilarity matrix from the data. For example, if you are dealing with binary attributes, you can create a contingency table where a is the number of features that equal 1 for both items x and y; b is the number of features that equal 1 for item x but equal 0 for object y; c is the number of features that equal 0 for item x but equal 1 for item y; and d is the number of features that equal 0 for both items x and y. The total number of features is z, where $z = a + b + c + d$ (see Table 7-5).

Table 7-5. *Contingency Table*

x	y			
		1	0	sum
	1	a	b	a+b
	0	c	d	c+d
	sum	a+c	b+d	z =(a+b+c+d)

You need to understand a couple of terms for calculating the dissimilarities.

Symmetric Binary Variable – Both levels of the variable have equal weight. For example, heads and tails of a coin.

Asymmetric Binary Variable – The two levels do not have the same importance. For example, with the results of a medical test, a positive may have a higher weightage than the negative occurrence.

Depending on the binary variable type, your dissimilarity calculations will vary. Symmetric Binary dissimilarity can be calculated using

$$d(x,y)= (b+c) / (a+b+c+d)$$

Asymmetric Binary dissimilarity can be calculated using

$$d(x,y)= (b+c) / (a+b+c)$$

Observe in the preceding equation that *d* is discarded in the denominator; this is because negative occurrence has little to no importance. Using this dissimilarity metric, you can calculate the similarity distance between *x* and *y* using the following formula:

$$\text{Similarity}(x,y) = 1 - d(x,y)$$
$$\text{Similarity}(x,y) = 1 - ((b+c) / (a+b+c))$$
$$\text{Similarity}(x,y) = c/(a+b+c)$$

This similarity measure is called *Jaccard's distance*. This can be used to determine the similarity between binary variables. But what happens to nominal variables? One workaround for nominal variables is to recode them to multiple binary variables. The other technique is to use a simple matching coefficient. It is calculated as follows:

Simple matching coefficient = number of matching attributes / number of attributes

In the case of a binary variable, this coefficient is similar to symmetric binary dissimilarity. The preceding coefficient gives the dissimilarity index. Similarity distance is calculated by subtracting this coefficient from 1:

Simple matching distance = 1 - Simple matching coefficient

We now demonstrate Jaccard's distance by continuing with the example used for Euclidean distances. Say a car dealership successfully made a deal with three respondents who owned three cars of different makes and models. The car dealership wants to categorize the cars of these respondents based on performance. They perform different tests on these cars to determine the similarity of the current performance. Failed tests are represented by *Y* and passed tests are represented by *N*. We have converted them to 1 and 0 respectively for convenience here (see Table 7-6).

Table 7-6. *Jaccard's Demonstration*

	Climatic	Mechanical	Pressure Impulse	Emissions Bench	Cat Aging	SHED
Car 1	1	0	1	0	0	0
Car 2	1	0	1	0	1	0
Car 3	1	1	0	0	0	0

You can calculate the dissimilarity between the cars using the following formula:

$$d(car1, car2) = 1+0 / (2+1+0)$$
$$d(car1, car2) = 0.33$$

How did we arrive at these numbers? Well, using the contingency table, we counted the number of *b*'s and *c*'s, as represented in Table 7-6. For Car 1 and Car 2, Cat Aging accounts to *b*. There is no *c* value here. Also note this is asymmetric binary similarity, so we would not account for the value of *d* in the denominator. From climatic and pressure impulse tests, we know the count of *a* is 2. To calculate the similarity, you need to subtract this index from 1:

$$Similarity(car1, car2) = 1 - 0.33$$
$$Similarity(car1, car2) = 0.67$$

Similarly, you can calculate the differences between other observations and create a similarity matrix.

Types of Clustering

Broadly, clustering can be categorized into hierarchical and non-hierarchical methods. This section focuses on explaining the differences between these types of clustering and on the algorithms available for each type of clustering. As the name suggests, hierarchical clustering tries to build a hierarchy of clusters in its routine to finalize the solution. Hierarchical clustering is further divided into agglomerative and divisive approaches(see Figure 7-3).

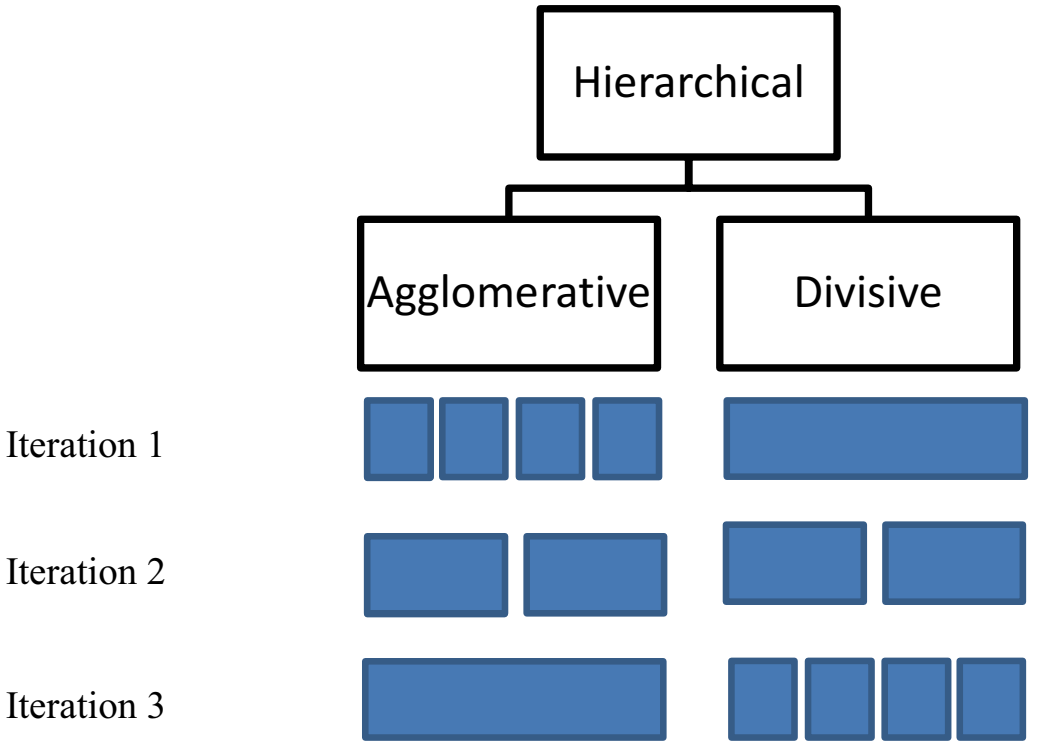


Figure 7-3. *Types of hierarchical clustering*

Agglomerative clustering: In this type of clustering, each observation starts in its own cluster and gradually works its way up to a single cluster. This approach is also referred to as *bottom-up clustering.*

Divisive clustering: In this type of clustering, all observations begin in a single cluster and are recursively divided into smaller clusters in each iteration. This approach is also referred to as *top-down clustering.*

Figure 7-3 depicts both approaches pictorially. We use selected data from Table 7-1 to demonstrate the agglomerative example. We also calculated the Euclidean distance for all these points (see Table 7-8). All pair-wise distances are represented in Table 7-7.

Table 7-7. *Agglomerative Example*

Respondent	Safety	Fuel Efficiency
1	10	10
2	9	9
6	10	1
7	9	1.5
11	1	10
12	1.5	9

Table 7-8. *Euclidean Distances of Selected Respondents*

	1	2	6	7	11	12
1	0.00	1.41	9.00	8.56	9.00	8.56
2	1.41	0.00	8.06	7.50	8.06	7.50
6	9.00	8.06	0.00	1.12	12.73	11.67
7	8.56	7.50	1.12	0.00	11.67	10.67
11	9.00	8.06	12.73	11.67	0.00	1.12
12	8.56	7.50	11.67	10.61	1.12	0.00

Note that this is a symmetric matrix; you can use either half to determine your clusters. When you start agglomerative clustering, all the datapoints are in their individual clusters. Based on the preceding data, the smallest distance observed between two respondents here is 1.12 for (6,7) and (11,12). There is a tie, so how will the algorithm decide which pairs should be fused? There are built-in rules to break this tie, and let's say it arbitrarily picks (6,7). In the first step, you would cluster them together and then calculate the overall heterogeneity measure, which here equals 1.12. Note that the heterogeneity measure as you begin the clustering is 0 because all the datapoints are

in their own individual clusters. Once you fuse 6 and 7 together, the next pair that gets fused is (11,12). The heterogeneity measure remains the same because the pair-wise distance between the points is still 1.12. In the next step, (1,2) will be fused together since they have the least pair-wise distance of 1.41. Now the average heterogeneity will go up to 1.21. This is calculated by taking (1.12+1.12+1.41)/3. These steps are repeated until all the observations fall into a single cluster.

You may wonder, how would you choose the right number of clusters? One way to identify this is to determine the break/fluctuation in the heterogeneity measure. Figure 7-4 gives you a visual depiction of how the algorithm works. Table 7-9 shows the steps.

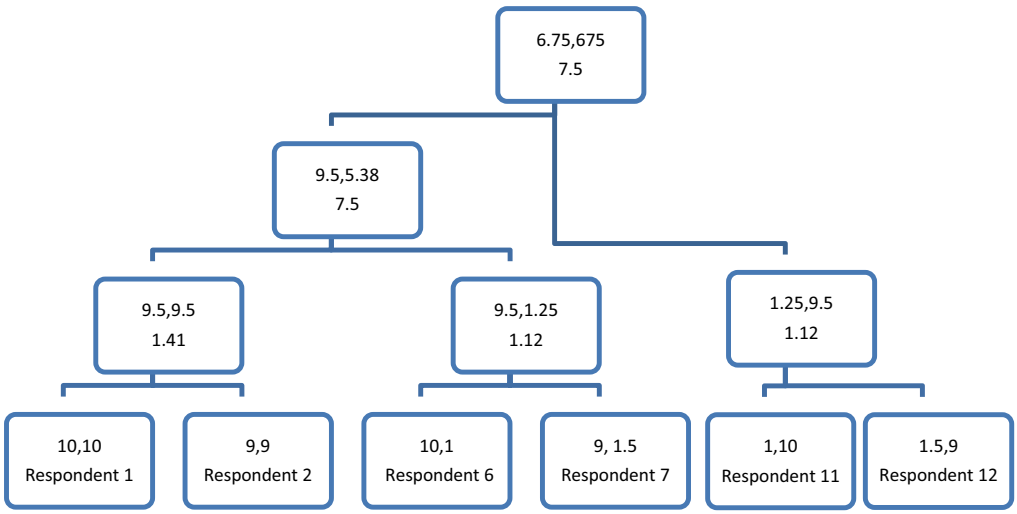


Figure 7-4. *Dendrogram: Hierarchical clustering based on single linkage. Centroids and Euclidean distances are represented*

Table 7-9. *Clustering Steps*

Step	Clusters	Heterogeneity
0	1 2 6 7 11 12	0
1	1 2 (**6 7**) 11 12	1.12
2	1 2 (**6 7**) (**11 12**)	1.12
3	(**1 2**) (**6 7**) (**11 12**)	1.21
4	(**1 2 6 7**) (11 12)	3.84
5	(**1 2 6 7 11 12**)	5.04

Observing the preceding heterogeneity measure, you can observe there is break in the measure from Step 3 to Step 4 and from Step 4 to Step 5. You can arrive at two possible solutions here, either a three-cluster or a four-cluster solution.

The method we used here is called *single linkage*. It looks at the nearest neighbor in determining the cluster measures and heterogeneity. Other methods available for looking at similarities include the following:

- **Complete linkage:** This uses the maximum distance between clusters to determine the cluster distances, instead of using the minimum. Figure 7-5 shows how the cluster attributes and distances vary.

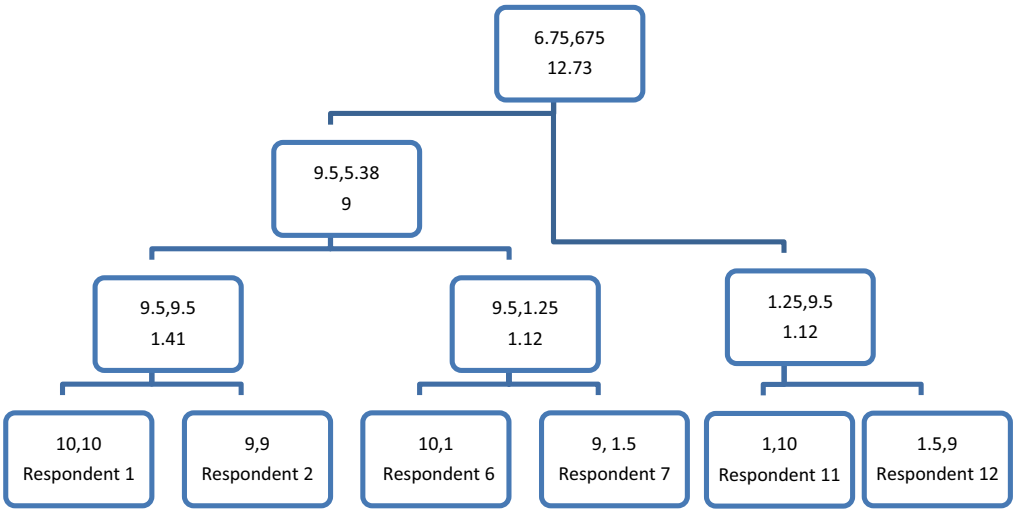


Figure 7-5. *Dendrogram: Hierarchical clustering based on complete linkage. Centroids and Euclidean distances are represented*

- **Average linkage:** This uses the pair-wise average distance between clusters to determine the cluster distances. Figure 7-6 shows how the cluster distances vary.

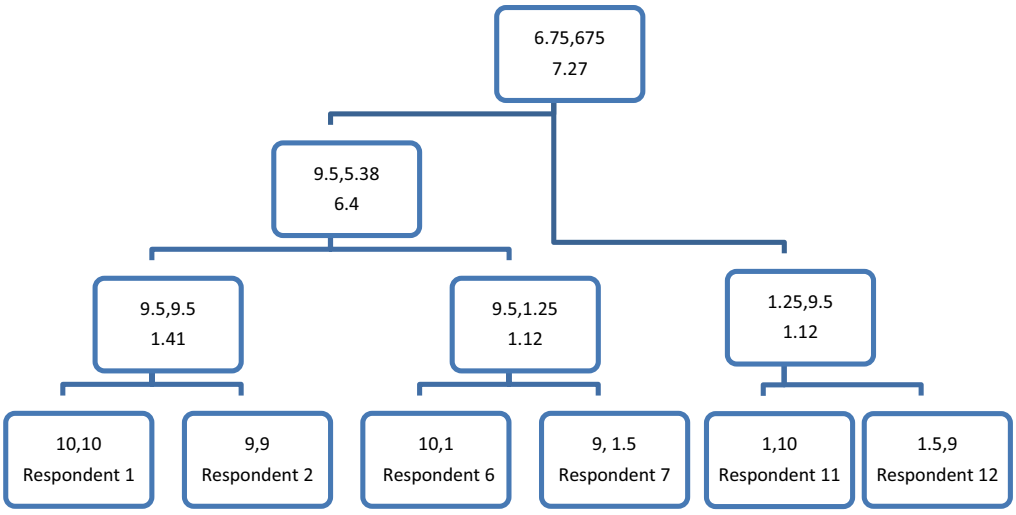


Figure 7-6. *Dendrogram: Hierarchical clustering based on average linkage. Centroids and Euclidean distances represented*

- **Centroid linkage:** This method uses the distance between the cluster centers to determine the cluster distances in the iterative steps.
- **Ward's method:** This method is based upon variance. It is based upon the principle of minimizing the cluster variance between observations within the cluster and maximizing the cluster variances across the clusters.

Although hierarchical clustering offers the advantage of not having to pick the initial number of clusters, it is a complex algorithm to implement, especially for a big data environment. It has a time and space complexity order of $O(n^2\log(n))$, where n is the number of observations. This can slow operations. On the brighter side, it can explain how the observations are grouped.

Bisecting k-Means

Bisecting k-means is a hybrid approach that combines aspects of hierarchical divisive clustering and non-hierarchical clustering. We introduce k-means in the next section, which is one of the more popular non-hierarchical clustering techniques. Bisecting k-means is similar to k-means in predefining the number of clusters. You may wonder how you can set the number of clusters without any prior knowledge of the data.

There are the techniques you saw earlier, like the heterogeneity measures, and other distance metrics you can use to determine the clusters if you run the algorithm for a few iterations.

Let's see how this bisecting k-means algorithm works in theory. Since this is a divisive algorithm, it starts at the top with a single cluster and then divides data iteratively. First, you need to set the required number of clusters.

As we start, all the data exists in a single cluster (see Figure 7-7). We split the data into two clusters (see Figure 7-8).

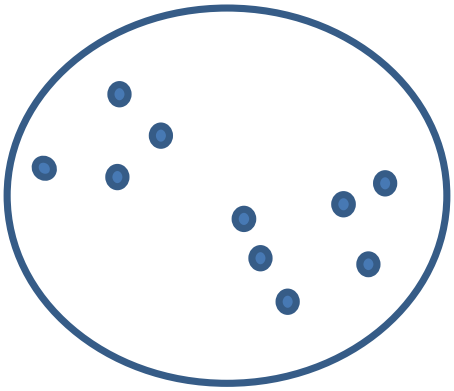


Figure 7-7. *Step 1: Bisecting k-means*

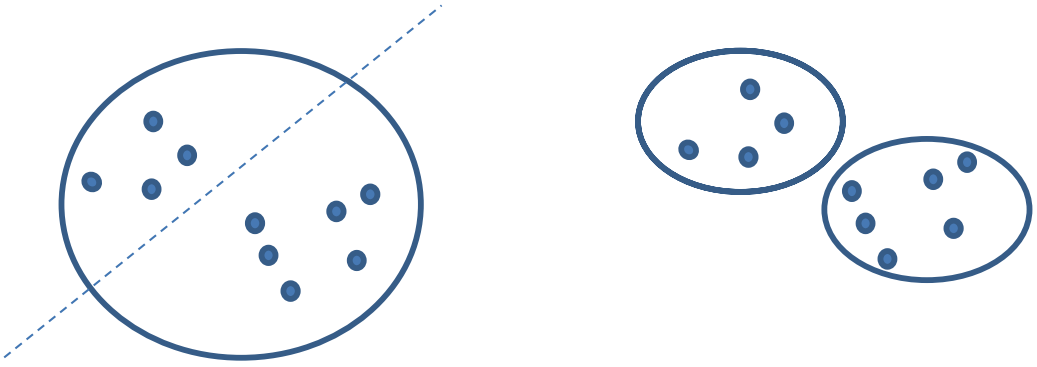


Figure 7-8. *Step 2: Bisecting k-means*

These first two clusters are divided using k-means. As the clusters are split, the centroid of each cluster is calculated. Then, the sum of the square of the distance measure between all the points and the centroid is calculated. This is called the sum of

squared distances/errors (SSE). The higher the SSE, the greater the heterogeneity of the cluster. In the next step, we choose the cluster with the highest SSE among the clusters and divide it into two clusters (see Figure 7-9).

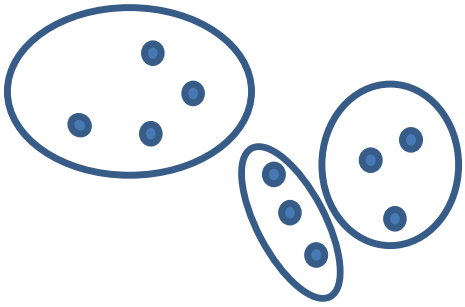


Figure 7-9. *Step 3: Bisecting k-means*

In the next step, we look at the SSE of the three clusters and choose the one with the highest SSE to bisect. We repeat the process until we arrive at the desired number of clusters. This process's low computation time is a great advantage over other processes. Bisecting k-means also produces clusters of similar sizes, unlike k-means, whose sizes vary greatly.

PySpark doesn't have an implementation of agglomerative hierarchical clustering as of version 3.0. There is a divisive hierarchical clustering method (top-down) approach using the `BisectingKMeans` function. For demonstration purposes, we use the credit card data that is sourced from Kaggle (`https://www.kaggle.com/arjunbhasin2013/ccdata`) (see Table 7-10).

Table 7-10. *Metadata Information of the Credit Card Dataset*

Column	Definition
custid	Identification of credit card holder (categorical)
balance	Amount left in their account to make purchases
balancefrequency	How frequently the balance is updated; a score between 0 and 1 (1 = frequently updated, 0 = not frequently updated)
purchases	Amount of purchases made from account
oneoffpurchases	Maximum purchase amount done in one go
installmentspurchases	Amount of purchase done in installments
cashadvance	Cash in advance given by the user
purchasesfrequency	How frequently purchases are being made; a score between 0 and 1 (1 = frequently purchased, 0 = not frequently purchased)
oneoffpurchasesfrequency	How frequently purchases happen in one go (1 = frequently purchased, 0 = not frequently purchased)
purchasesinstallmentsfrequency	How frequently purchases in installments are done (1 = frequently done, 0 = not frequently done)
cashadvancefrequency	How frequently cash in advance is paid
cashadvancetrx	Number of transactions made with "cash in advance"
purchasestrx	Number of purchase transactions made
creditlimit	Limit of credit card for user
payments	Amount of payment done by user
minimum_payments	Minimum amount of payments made by user
prcfullpayment	Percentage of full payment paid by user
tenure	Tenure of credit card service for user

This dataset can't be used in its raw form. We know from the descriptions that these data elements have different scales. We have to normalize them and form a feature vector before we can apply the clustering technique.

```
# Read data
file_location = "cluster_data.csv"
file_type = "csv"
infer_schema = "false"
first_row_is_header = "true"
df = spark.read.format(file_type)\
.option("inferSchema", infer_schema)\
.option("header", first_row_is_header)\
.load(file_location)
# Print metadata
df.printSchema()
# Casting multiple variables
from pyspark.sql.types import *
#Identifying and assigning lists of variables
float_vars=list(set(df.columns) - set(['CUST_ID']))
for column in float_vars:
      df=df.withColumn(column,df[column].cast(FloatType()))
# Imputing data
from pyspark.ml.feature import Imputer
# Shortlisting variables where mean imputation is required
input_cols=list(set(df.columns) - set(['CUST_ID']))
# Defining the imputer function
imputer = Imputer(
    inputCols=input_cols,
    outputCols=["{}_imputed".format(c) for c in input_cols])
# Applying the transformation
df_imputed=imputer.fit(df).transform(df)
# Dropping the original columns as we created the _imputed columns
df_imputed=df_imputed.drop(*input_cols)
# Renaming the input columns to original columns for consistency
new_column_name_list= list(map(lambda x: x.replace("_imputed", ""),
df.columns))
df_imputed = df_imputed.toDF(*new_column_name_list)
# Data Preparation
from pyspark.ml.feature import VectorAssembler
```

```
from pyspark.ml.feature import Normalizer
from pyspark.ml import Pipeline
# Listing the variables that are not required in the segmentation analysis
ignore = ['CUST_ID']
# creating vector of all features
assembler = VectorAssembler(inputCols=[x for x in df.columns if x not in
ignore], outputCol='features')
# creating the normalization for all features for scaling betwen 0 to 1
normalizer = Normalizer(inputCol="features",
outputCol="normFeatures", p=1.0)
# Defining the pipeline
pipeline = Pipeline(stages=[assembler, normalizer])
# Fitting the pipeline
transformations=pipeline.fit(df_imputed)
# Applying the transformation
df_updated = transformations.transform(df_imputed)
# Building the model
from pyspark.ml.clustering import BisectingKMeans
from pyspark.ml.evaluation import ClusteringEvaluator
# Trains a bisecting k-means model.
bkm = BisectingKMeans().setK(2).setSeed(1)
model = bkm.fit(df_updated.select('normFeatures').withColumnRenamed(
'normFeatures','features'))
# Make predictions
predictions = model.transform(df_updated.select('normFeatures').withColumn
Renamed('normFeatures','features'))
# Evaluate clustering by computing Silhouette score
evaluator = ClusteringEvaluator()
silhouette = evaluator.evaluate(predictions)
print("Silhouette with squared euclidean distance = " + str(silhouette))
# Shows the result.
print("Cluster Centers: ")
centers = model.clusterCenters()
for center in centers:
    print(center)
```

We use the Silhouette score to determine the optimal number of clusters. This is one of the best available metrics to determine the right number of clusters. This metric is calculated for each sample in the dataset. It is calculated using the following formula:

Silhouette Coefficient = (x-y)/ max(x,y)

The *x* is the inter-cluster distance and *y* is the intra-cluster distance. The coefficient can vary between -1 and 1. Values close to 1 define a clear separation and are more desirable. Let's now calculate the Silhouette coefficient for different numbers of clusters and plot it to determine the right cluster number (see Figure 7-10).

```
# Iterations
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
sil_coeff=[]
num_clusters=[]
for iter in range(2,8):
    bkm = BisectingKMeans().setK(iter).setSeed(1)
    model = bkm.fit(df_updated.select('normFeatures').withColumnRenamed(
    'normFeatures','features'))
    # Make predictions
    predictions = model.transform(df_updated.select('normFeatures').with
    ColumnRenamed('normFeatures','features'))
    # Evaluate clustering by computing Silhouette score
    evaluator = ClusteringEvaluator()
    silhouette = evaluator.evaluate(predictions)
    sil_coeff.append(silhouette)
    num_clusters.append(iter)
    print("Silhouette with squared euclidean distance for "+str(iter) +"
    cluster solution = " + str(silhouette))
# create a dataframe for visualization
df_viz=pd.DataFrame(zip(num_clusters,sil_coeff), columns=['num_
clusters','silhouette_score'])
sns.lineplot(x = "num_clusters", y = "silhouette_score", data=df_viz)
plt.title('Bisecting k-means : Silhouette scores')
plt.xticks(range(2, 8))
plt.show()
```

```
Silhouette with squared euclidean distance for 2 cluster solution = 0.49855848955650006
Silhouette with squared euclidean distance for 3 cluster solution = 0.47032306895494647
Silhouette with squared euclidean distance for 4 cluster solution = 0.36841230437497996
Silhouette with squared euclidean distance for 5 cluster solution = 0.35979911858435865
Silhouette with squared euclidean distance for 6 cluster solution = 0.3787940202378902
Silhouette with squared euclidean distance for 7 cluster solution = 0.3639517765678912
```

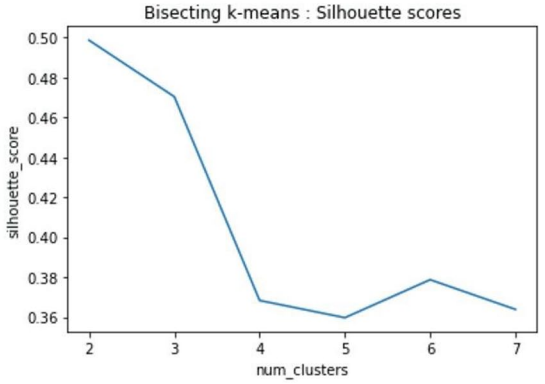


Figure 7-10. *Silhouette score changing over multiple iterations using the bisecting k-means with an increasing number of clusters*

K-means

By far, k-means is one of the most popular clustering techniques since it scales relatively easily with massive data. Most other techniques are computationally expensive when the data is large. One of the advantages and sometimes a disadvantage with k-means is that you can dictate the number of clusters that you want. For demonstration purposes, we use the same car survey data from Table 7-1. We set the number of clusters to be three. The algorithm assigns three random initialization points in space, as shown in Figure 7-11. These points represent initial cluster centers.

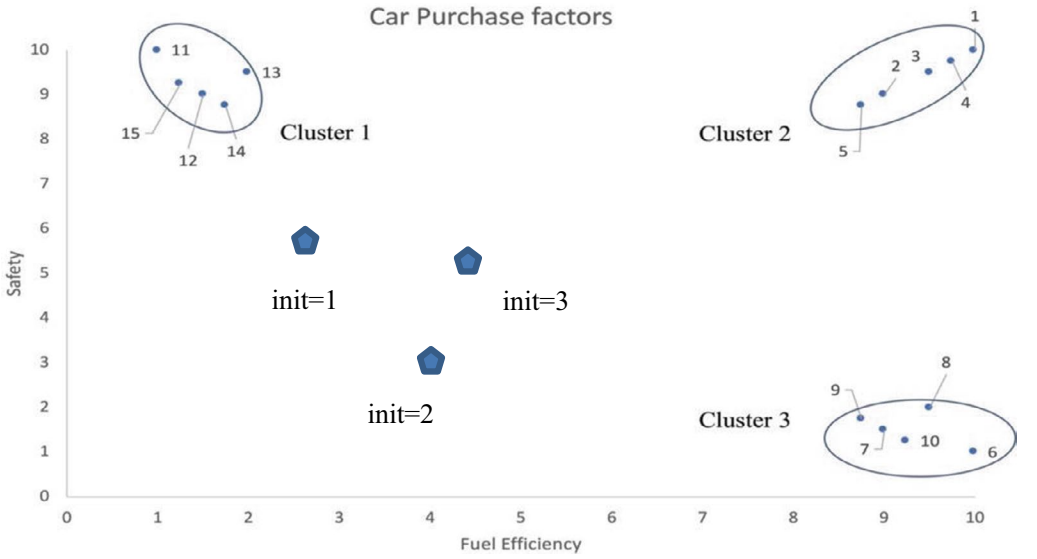


Figure 7-11. *K-means demonstration*

In the second step, the distances of all points from the three initialization-point cluster centers are calculated. Based on the minimum distance to the cluster centers, all points are assigned to one of the clusters. Centroids of new cluster points are calculated to find the new cluster center. Again, the distances of all points from the new cluster centers are calculated. Based on the minimum distance to cluster centers, points are reassigned. This process repeats until there is no change in the points in a cluster.

You may wonder, how do you identify the right number of clusters in k-means? Well, you know the answer from hierarchical clustering. You use the cluster variances to determine the right number of clusters. You run the clustering from 0 to *n*, plotting the change in the cluster variances. This plot is called an *elbow plot*. Whenever you observe an elbow or a sharp change in the plot, you can use that point as the optimal number of clusters. The goal is identifying the smallest value of *k* that has the lowest sum of squares (SSE).

We will make use of the same credit card dataset to implement k-means in PySpark.

EXERCISE 7-1: SEGMENTATION

Question 1: Observe the differences between hierarchical and non-hierarchical clustering by setting the same number of clusters.

- Are the cluster profiles and sizes similar?

```
# Import Sparksession
from pyspark.sql import SparkSession
spark=SparkSession.builder.appName("Clustering").getOrCreate()
# Print PySpark and Python versions
import sys
print('Python version: '+sys.version)
print('Spark version: '+spark.version)
# Read data
file_location = "cluster_data.csv"
file_type = "csv"
infer_schema = "false"
first_row_is_header = "true"
df = spark.read.format(file_type)\
.option("inferSchema", infer_schema)\
.option("header", first_row_is_header)\
.load(file_location)
# Print metadata
df.printSchema()
#  Count data
df.count()
print('The total number of records in the credit card dataset are
'+str(df.count()))
# Cast multiple variables
from pyspark.sql.types import *
# Identify and assign lists of variables
float_vars=list(set(df.columns) - set(['CUST_ID']))
for column in float_vars:
    df=df.withColumn(column,df[column].cast(FloatType()))
```

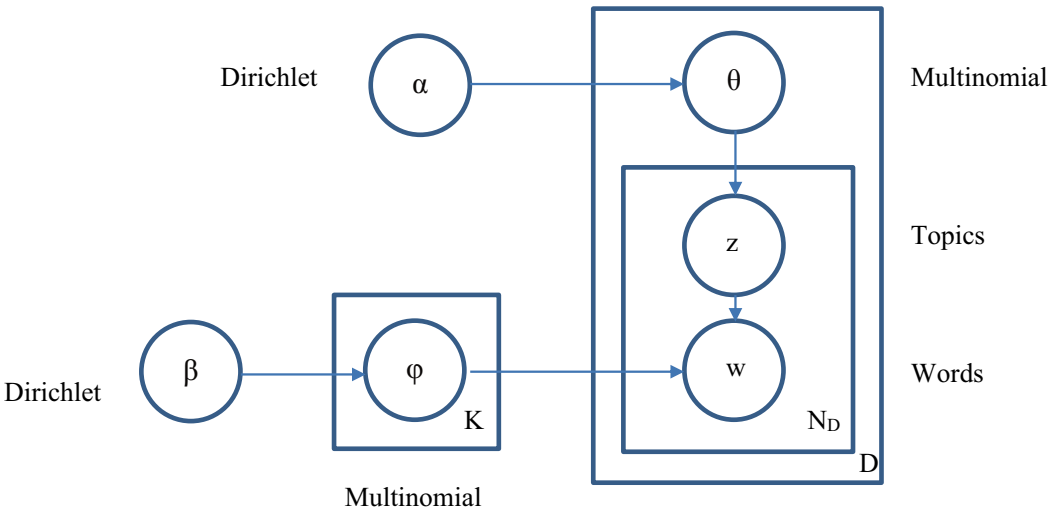
```
from pyspark.ml.feature import Imputer
# Shortlist variables where mean imputation is required
input_cols=list(set(df.columns) - set(['CUST_ID']))
# Define the imputer function
imputer = Imputer(
    inputCols=input_cols,
    outputCols=["{}_imputed".format(c) for c in input_cols])
# Apply the transformation
df_imputed=imputer.fit(df).transform(df)
# Drop the original columns as we created the _imputed columns
df_imputed=df_imputed.drop(*input_cols)
# Rename the input columns to original columns for consistency
new_column_name_list= list(map(lambda x: x.replace("_imputed", ""),
df.columns))
df_imputed = df_imputed.toDF(*new_column_name_list)
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import Normalizer
from pyspark.ml import Pipeline
# List the variables that are not required in the segmentation analysis
ignore = ['CUST_ID']
# create vector of all features
assembler = VectorAssembler(inputCols=[x for x in df.columns if x not in
ignore], outputCol='features')
# create the normalization for all features for scaling betwen 0 to 1
normalizer = Normalizer(inputCol="features",
outputCol="normFeatures", p=1.0)
# Define the pipeline
pipeline = Pipeline(stages=[assembler, normalizer])
# Fit the pipeline
transformations=pipeline.fit(df_imputed)
# Apply the transformation
df_updated = transformations.transform(df_imputed)
from pyspark.ml.clustering import KMeans
from pyspark.ml.evaluation import ClusteringEvaluator
```

```
# Train a k-means model.
kmeans = KMeans().setK(2).setSeed(1003)
model = kmeans.fit(df_updated.select('normFeatures').withColumnRenamed(
'normFeatures','features'))
# Make predictions
predictions = model.transform(df_updated.select('normFeatures').withColumn
Renamed('normFeatures','features'))
# Evaluate clustering by computing Silhouette score
evaluator = ClusteringEvaluator()
silhouette = evaluator.evaluate(predictions)
print("Silhouette with squared euclidean distance = " + str(silhouette))
# Show the result.
centers = model.clusterCenters()
print("Cluster Centers: ")
for center in centers:
    print(center)
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
sil_coeff=[]
num_clusters=[]
for iter in range(2,8):
    kmeans = KMeans().setK(iter).setSeed(1003)
    model = kmeans.fit(df_updated.select('normFeatures').withColumnRenamed(
    'normFeatures','features'))
    # Make predictions
    predictions = model.transform(df_updated.select('normFeatures').with
    ColumnRenamed('normFeatures','features'))
    # Evaluate clustering by computing Silhouette score
    evaluator = ClusteringEvaluator()
    silhouette = evaluator.evaluate(predictions)
    sil_coeff.append(silhouette)
    num_clusters.append(iter)
    print("Silhouette with squared euclidean distance for "+str(iter) +"
    cluster solution = " + str(silhouette))
```

```
df_viz=pd.DataFrame(zip(num_clusters,sil_coeff), columns=['num_clusters',
'silhouette_score'])
sns.lineplot(x = "num_clusters", y = "silhouette_score", data=df_viz)
plt.title('k-means : Silhouette scores')
plt.xticks(range(2, 8))
plt.show()
```

Latent Dirichlet Allocation (LDA)

Latent Dirichlet Allocation (LDA) can help you identify abstract topics in a collection of documents using statistical models. Introduced in 2003, LDA has been used in many applications beyond text analysis, such as sentiment analysis and object localization for images. To explain LDA, we start with a simple diagram illustration (see Figure 7-12).



$$P(\boldsymbol{W}, \boldsymbol{Z}, \boldsymbol{\theta}, \boldsymbol{\varphi}; \alpha, \beta) = \prod_{i=1}^{K} P(\varphi_i; \beta) \prod_{j=1}^{M} P(\theta_j; \alpha) \prod_{t=1}^{N} P(Z_{j,t} \mid \theta_j) P(W_{j,t} \mid \varphi_{Z_{j,t}})$$

Words Topics Dirichlet

Topics Words Multinomial

Figure 7-12. *Latent Dirichlet Allocation demonstration*

At first glance, the diagram and the formula can be overwhelming. Let's break this down and understand the components. First, what is a Dirichlet? Dirichlet distribution is a family of continuous multivariate probability distributions parameterized by a vector of positive reals. It is a multivariate generalization of the beta. $\propto$ and β in Figure 712 are the Dirichlet distributions. θ and φ are multinomial distributions. Using these distributions, we create words and topics, which in turn can create documents.

You can think of LDA as an engine that spits out documents. In the left side of the equation, we have the probability of a document appearing. By adjusting the factors on the right, we can expect the generation of different flavors of documents with topics and words. The first two components of the equation are Dirichlet distributions. Words (β) can be a simplex of an order n depending upon the number of keywords. Topics ($\propto$) can be a simplex of an order less than or equal to n.

Consider Dirichlet visually. Say you have two topics, sports and education. You can represent them visually as the Dirichlet in Figure 7-13. You can add another topic, environment; its representation is also shown. If there are four topics, remember it will be a simplex of order 3, which is a tetrahedron and not a square.

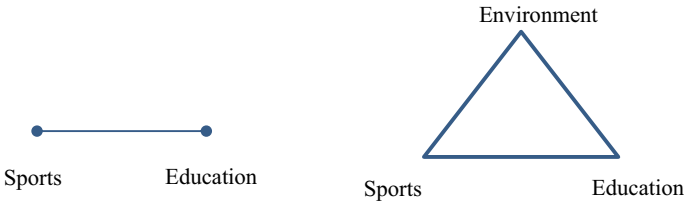


Figure 7-13. *Dirichlet representation*

If documents contain sports, they will be represented toward the sports side, and documents for education will be represented toward education. This is how the distributions will play out with different values of $\propto$ (see Figure 7-14).

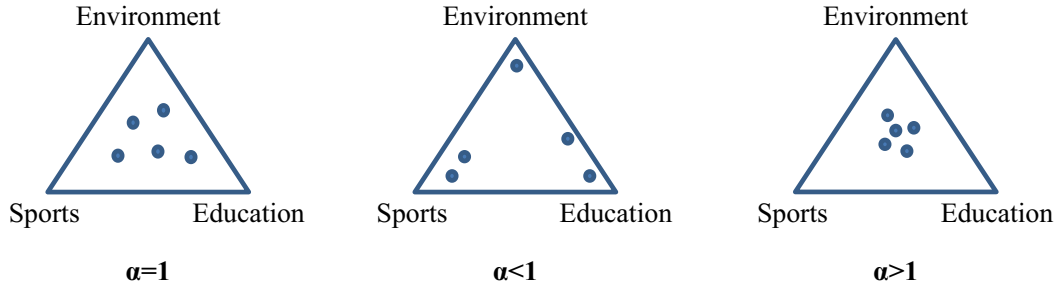


Figure 7-14. *Variations of Dirichlet distribution*

If the value of $\propto$ is equal to 1, there is uniform distribution of topics in the articles. If the value of $\propto$ is less than 1, there is good separation of topics and the documents are dispersed toward the corners. If the value of $\propto$ is greater than 1, all the documents are clustered in the center and there is no good separation of topics. The contents of most articles tend to follow the $\propto$<1 distribution. The next question is, how about words? We repeat the same notion for the words. In Figure 7-15, we used four words: league, planet, school, and goal. We place the words in the corners and points as the topics.

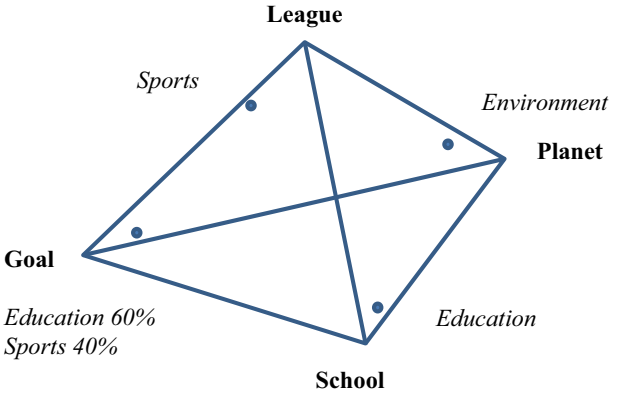


Figure 7-15. *Topics of words*

We can now put these into equation perspective and demonstrate the generation of a single document from this LDA. For this demonstration, we break down the formula into four factors, as shown in Figure 7-16.

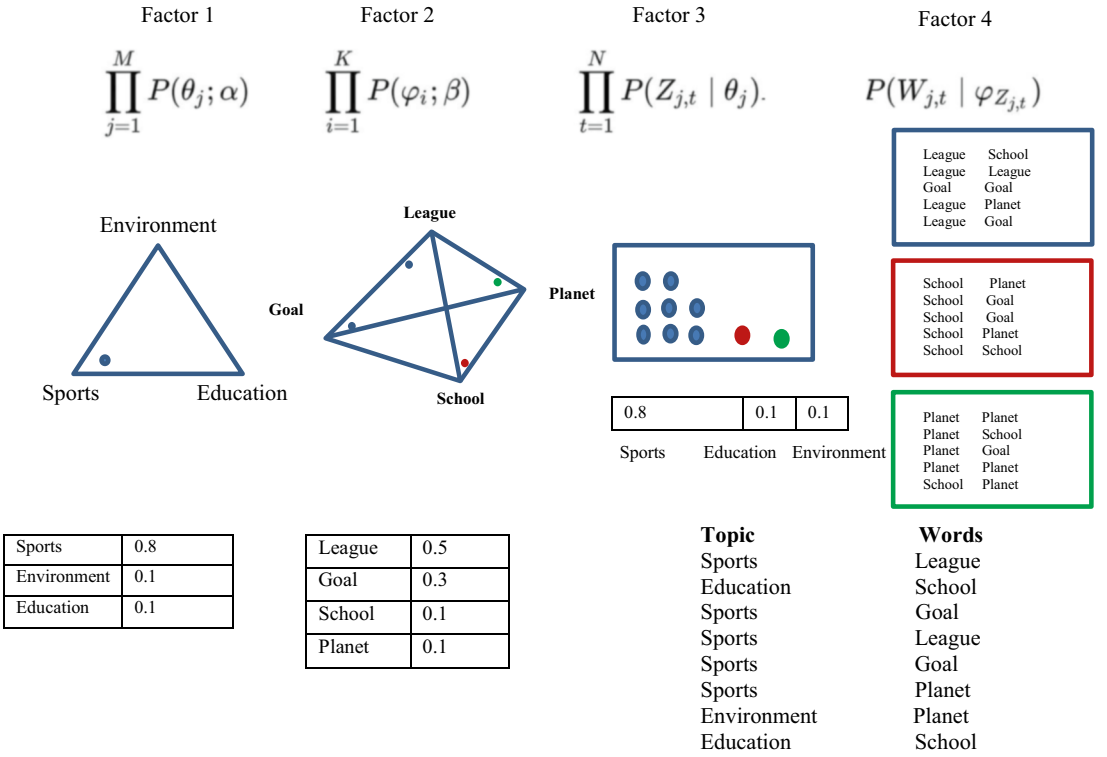


Figure 7-16. *Formula breakdown*

The first factor is the simplex of order 2, where the corners contain the topics, and we are picking one random point to be one of the documents that we are going to create. Note that this point is oriented toward the sports corner. We can represent this point as a document containing approximately 80 percent sports, 10 percent environment, and 10 percent education. We use these percentages to create multinomial distributions in factor 3. As you can see in Figure 7-16, we mimicked the same percentage distribution in points within the box for sports, education, and environment. If we pick a random topic from the topic box with replacement multiple times, we can expect the topic distribution to be as shown in Figure 7-16 under factor 3.

Now we want to find the words corresponding to the topics. For this, we pick factor 2, which has words in the corners and topics represented at the points. We quantify the percentage of words associated with sports based on their position in the tetrahedron. This is represented in the table below factor 2. This corresponding multinomial distribution is represented in factor 4 in the first box. Similar multinomial distributions can be obtained for the remaining topics of education and environment based on the factor 2 Dirichlet distributions of education and environment.

Now, we go back to factor 3, topic distributions. We select the first topic, which is sports. We look up the corresponding multinomial distribution of words from factor 4 and select a random word from the first box, which happens to be *league*. We then move to the second instance of topic, which is education. We then use the second box, which represents education, from factor 4 and select a random word. Let's say this random word is *school*. We repeat this process for the remaining topics to identify the corresponding words.

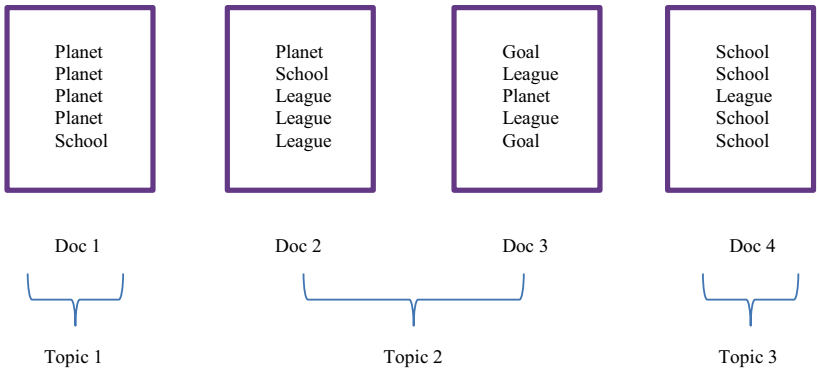
All the words identified from the topics stacked together will create our document. This machine-generated document may not be meaningful and is a combination of different words. That is acceptable, and we repeat this process multiple times, changing the settings of the four factors to generate multiple documents. We verify the similarity of these machine-generated documents with the input documents. There is a low probability that the original documents are similar to the machine-generated documents in multiple iterations. As we repeat this process multiple times, we observe how the similarity changes with each iteration and select the settings that maximize the similarity between the machine-generated documents and input documents.

LDA Implementation

You may now wonder how can determine the number of topics when you are presented with a corpus of documents. The answer is through hyperparameter settings. In LDA, the number of topics is also a hyperparameter. Results vary according to the number of topics selected. When you present a corpus of documents to the computer, it won't understand any interpretation as humans do. But computers are good at recognizing the frequency of word occurrences in a document. You can also verify how these words occur in different documents across the corpus. Using this information, you can tag words to topics. Two major properties drive this process, as follows:

- Each article predominantly contains a single topic
- Each word is primarily associated with a single topic

If we break down the preceding properties, taking an example of the four documents as shown in Figure 7-17, we will find documents that contain a mix of words; usually there is one dominant topic that exists in the documents. We can quantitatively determine the number of occurrences of words and then tag them to different topics accordingly. Say we have chosen three topics; we will classify these four documents into Topic 1, Topic 2, and Topic 3.



Word	Topic	Occurrences
Planet	1	4
	2	2
	3	0

Word	Topic	Occurrences
School	1	1
	2	1
	3	4

Word	Topic	Occurrences
Goal	1	0
	2	2
	3	0

Word	Topic	Occurrences
League	1	0
	2	5
	3	1

Figure 7-17. *Example corpus*

The second property is demonstrated in the word occurrences by topic. Although there are multiple instances where one word is associated with multiple topics, each word is strongly oriented toward a single topic. For simplicity, assume the majority drives the word assignment to the topic. Then we will have the data shown in Table 7-11.

Table 7-11. *Topic Assignment*

Word	Topic
Planet	Topic 1
Goal	Topic 2
League	Topic 2
School	Topic 3

If there are some commonly occurring words, such as *the, is,* and *any,* they are eliminated using the natural language processing technique of *term frequency – inverse document frequency* (tf–idf). This is a technique used to identify the most important words in a corpus. The tf–idf value increases proportionally to the number of times a word appears in the document and is offset by the number of documents in the corpus that contain the word, which helps to adjust for the fact that some words appear more frequently in general.

We can condense the problem statement as classifying all the words into the given/chosen number of topics, retaining a single topic for each document if possible and each word into a single topic if possible. This is done through *Gibbs sampling*. What we are trying to do is figure out the latent variables that describe what's going on in a topic model. We are assuming a collection of latent variables that completely describe how we are going from topics to allocations to assignments to generate our complete document.

Gibbs sampling allows us to organize all the words one by one with the right topics. This is done in a couple of steps. The first step involves a word search within the document, and the second step involves a word search across documents. Using Figure 7-17, from Document 1, we pick the first word, *planet*. We will not assign this word to any topic, but we count how the other words in the document are assigned to different topics. We have three planets and one school word. These three planet words are assigned to Topic 1, and school is assigned to Topic 3 based on Table 7-11. Now we will look at how the word planet is occurring across different topics (Table 7-12).

Table 7-12. *Topic Occurrences*

Topic	Topic Word Assignments	Occurrences	Estimates
Topic 1	How much of Topic 1 is in Doc 1	3	$2 + \alpha$
	How much of Planet is in Topic 1	4	$4 + \beta$
Topic 2	How much of Topic 2 is in Doc 1	0	$0 + \alpha$
	How much of Planet is in Topic 2	2	$2 + \beta$
Topic 3	How much of Topic 3 is in Doc 1	1	$1 + \alpha$
	How much of Planet is in Topic 3	0	$0 +$

Using the occurrences, we can calculate the probability of a word belonging to a topic. If we just use the occurrences alone for each topic, we may run into the problem of estimating 0, giving no chance to other topics for a selected word. To overcome this, we add small adjustment terms α and β. If you recall the equation from Figure 7-16, α and β are Dirichlet distributions. We repeat this for all the words in each document multiple times until we maximize the two properties discussed earlier in the section. A key takeaway from Gibbs sampling is that you can reclassify words so that they are associated with a single topic if possible. As this process is done iteratively, the computer will help you identify the topics in each document and give you a list of words that define the topic. As a final step, we have to label these topics, observing the keywords.

We will use a real-world example to show how LDA can be implemented using PySpark. For this purpose, we use a dataset from Kaggle (https://www.kaggle.com/PromptCloudHQ/amazon-echo-dot-2-reviews-dataset). This dataset contains reviews of the Echo Dot, a smart speaker. It contains the following fields:

- Page URL
- Title
- Review Text
- Device Color
- User Verified
- Review Date
- Review Useful Count
- Configuration
- Rating
- Declaration Text (Example: Vine Voice, Top 100 reviewer, etc.)

We use the Review Text column for this modeling exercise.

```
# Import Sparksession
from pyspark.sql import SparkSession
spark=SparkSession.builder.appName("LDA").getOrCreate()
# Print PySpark and Python versions
import sys
print('Python version: '+sys.version)
```

```
print('Spark version: '+spark.version)
# Read data
file_location = "lda_data.csv"
file_type = "csv"
infer_schema = "false"
first_row_is_header = "true"
df = spark.read.format(file_type)\
.option("inferSchema", infer_schema)\
.option("header", first_row_is_header)\
.load(file_location)
# Print metadata
df.printSchema()
#  Count data
df.count()
print('The total number of records in the credit card dataset are
'+str(df.count()))
%%bash
pip install nltk
# Import appropriate libraries
from pyspark.sql.types import *
from pyspark.mllib.linalg import Vector, Vectors
from pyspark.ml.feature import CountVectorizer , IDF
from pyspark.mllib.clustering import LDA, LDAModel
from pyspark.mllib.linalg import Vectors as MLlibVectors
import re
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
```

In the preceding steps, we are reading the data, importing the necessary libraries, and exploring the metadata associated with the data.

```
# Extract review text from DataFrame and filter out None values
reviews = df.rdd.map(lambda x : x['Review Text']).filter(lambda x: x is
not None)
# Define English stopwords from NLTK
```

```
StopWords = stopwords.words("english")
tokens = reviews.map(lambda document: document.strip().lower())\
    .map( lambda document: re.split(" ", document)) \
    .map( lambda word: [x for x in word if x.isalpha()]) \
    .map( lambda word: [x for x in word if len(x) > 3] )\
    .map( lambda word: [x for x in word if x not in StopWords]).
zipWithIndex()
```

In the preceding steps, we change the DataFrame to RDD and filter out the empty reviews. Then, we load the stopwords from the NTLK library and use multiple lambda functions to strip extra spaces, change to lowercase, check alphanumerics, and remove any words less than three letters long, including stopwords. Finally, we add an index to each row to use them as identifiers later in the process.

```
# Convert the rdd to DataFrame
df_txts = spark.createDataFrame(tokens, ['list_of_words','index'])
# TF
cv = CountVectorizer(inputCol="list_of_words", outputCol="raw_features",
vocabSize=5000, minDF=10)
cvmodel = cv.fit(df_txts)
result_cv = cvmodel.transform(df_txts)
# IDF
idf = IDF(inputCol="raw_features", outputCol="features")
idfModel = idf.fit(result_cv)
result_tfidf = idfModel.transform(result_cv)
```

In the preceding steps, we are creating term frequency and inverse document frequency using the built-in functions. `CountVectorizer` helps convert the collection of text documents to vectors of token counts. IDF reweights features to lower values that appear frequently in a corpus.

We are using this data, which contains the words with high importance from each document, in the LDA model. We then print the top five words from each topic to understand the topic. Here we used ten topics, but is this the best? No, we will have to try different combinations to identify the right number. Ten is used here as an example. You can iterate the following over different combinations and observe which number gives you a meaningful topic collection.

```
num_topics = 10
max_iterations = 100
lda_model=LDA.train(result_tfidf.select("index", "features").rdd.
mapValues(MLlibVectors.fromML).map(list),k = num_topics, maxIterations =
max_iterations)
wordNumbers = 5
data_topics=lda_model.describeTopics(maxTermsPerTopic = wordNumbers)
vocabArray = cvmodel.vocabulary
topicIndices = spark.sparkContext.parallelize(data_tp)
def topic_render(topic):
    terms = topic[0]
    result = []
    for i in range(wordNumbers):
        term = vocabArray[terms[i]]
        result.append(term)
    return result
topics_final = topicIndices.map(lambda topic: topic_render(topic)).
collect()
for topic in range(len(topics_final)):
    print ("Topic" + str(topic) + ":")
    for term in topics_final[topic]:
        print (term)
    print ('\n')
```

Output:

```
Topic0:
works
good
speaker
sound
better
Topic1:
know
time
```

```
questions
playing
alexa
Topic2:
amazon
another
first
purchased
back
Topic3:
would
device
work
could
give
Topic4:
well
need
thing
pretty
loves
Topic5:
love
great
bought
little
much
Topic6:
home
weather
smart
voice
things
Topic7:
really
```

```
getting
without
wish
enjoy
Topic8:
music
echo
play
every
want
Topic9:
product
easy
still
alexa
using
```

Collaborative Filtering

If you have used or have been presented with any recommendation engines, the chances are highly likely that collaborative filtering was used under the hood. One of the advantages of collaborative filtering is that it doesn't need an extensive load of customer data except for historical preference on a set of items. One of the inherent assumptions is that future preferences are consistent with past preferences. There are two major categories of user preferences, explicit and implicit ratings. As the name suggests, *explicit ratings* are given by the user on an agreed service provider scale (e.g., 1 to 10 for IMDB). This presents a great opportunity, as we know how strongly a user likes or dislikes a product/content. *Implicit ratings,* on other hand, are measured through engagement activities (e.g., how many times a customer listened to a particular track, purchases, viewing habits, etc.).

Data in the real world is usually bimodal. It is created as a joint interaction between two types of entities. For example, a user rating a movie is affected by both user and movie characteristics. If you imagine a dataset with rows containing different users and columns containing their ratings of movies, classical clustering techniques would look at

how close the users are in the multi-dimensional space. We often ignore the relationship between the columns. Co-clustering aims to group objects by both rows and columns simultaneously based on the similarity of their pair-wise interactions. Co-clustering is powerful when we have sparse data. Data is sparse in most real-world cases. A viewer can watch a finite number of movies, so as we pick data across users and movies, data will be sparse, and this is one of the best examples of co-clustering. In the collaborative filtering lingo, they are called user-based collaborative filtering and item-based collaborative filtering.

User-based Collaborative Filtering

Say you have a x b matrix of ratings with users u_i, i =1 to n; and items p_j, j=1 to m; if you wanted to predict rating r_{ij} and if the user did not watch item j, you could calculate similarities between user i and all other users (see Figure 7-18). You could then select top similar users and take the weighted average of ratings from these top users with similarities as weights.

$$r_{ij} = \frac{\sum_k Similarities\left(u_i, u_k\right) r_{kj}}{number\ of\ ratings}$$

Again, you may have different types of raters, ranging from strict to lenient. To avoid the bias, you can use each user's average rating of all items when computing and add it back to the target.

$$r_{ij} = \bar{r}_i + \frac{\sum_k Similarities\left(u_i, u_k\right)\left(r_{kj} - \bar{r}_k\right)}{number\ of\ ratings}$$

There are other ways to calculate similarity, such as Pearson correlation and cosine similarity. All these metrics allow you to find the most similar users to your target user and are based on the principle that users are similar when they engage equally or purchase similar items.

Item-based Collaborative Filtering

For item-based collaborative filtering, you predict items that a user might prefer based on their taste using the information collected from other users having similar tastes or preferences. This approach offers stability but it is computationally intensive.

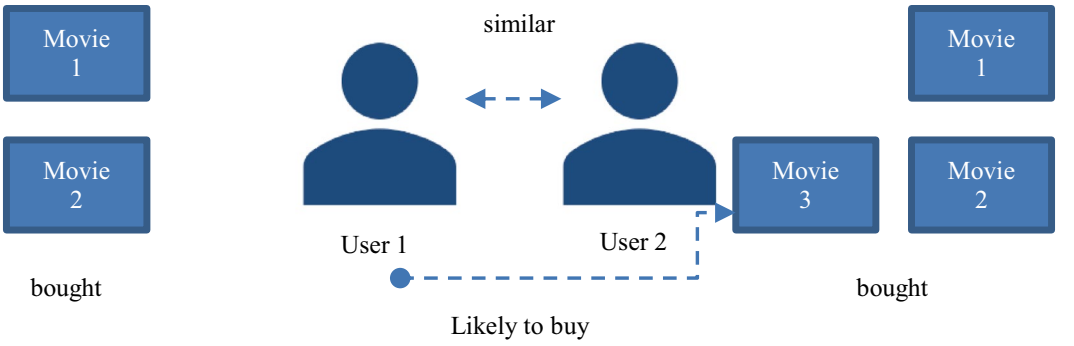


Figure 7-18. *User-based collaborative filtering*

Matrix Factorization

Co-clustering requires the use of matrix factorization. It is a method of decomposing an initial sparse matrix of user and item-based interactions into low-dimensional matrices with latent features that have lower sparsity. Latent features can be a broad categorization of items (e.g., fresh produce in grocery, genres in movies, and so on). Matrix factorizations give you an estimation of how much a movie fits into a category and how much a user fits into the same category. One of the greatest benefits of this method is even if there is no explicit or implicit rating of common items between two users, you can still find any similarities using the latent features that they share (see Figure 7-19).

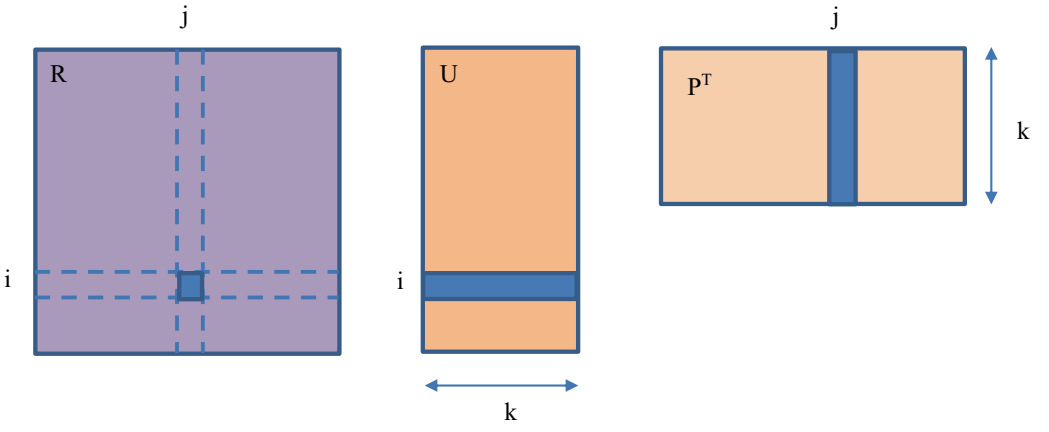


Figure 7-19. *Matrix factorization*

Matrix factorization will split the user and item matrix R into U (user based) and P (item based) such that $R \sim U \times P$, $U \in R^{m*k}$; $P \in R^{n*k}$. The k here is the rank of the factorization. Now you can treat this as an optimization process that approximates the original matrix R with U and P minimizing the cost function J:

$$J = \| R - U \times P^T \|_2 + \lambda \left(\| U \|_2 + \| P \|_2 \right)$$

The first term represents the mean square error distance between R and its approximation $U \times P^T$. The second term is a regularization term to prevent overfitting.

Optimization Using Alternating Least Squares (ALS)

ALS is used to minimize the loss function by iteratively updating half of the parameters while fixing the other half. This alternating approach avoids the need for gradient computation in the optimization step. It is an efficient solution to matrix factorization for implicit feedback. The costs of gradient computation can be extremely hard, as we assume unobserved entries to be 0. You will gain deeper insight as we discuss more about the cost function.

Based on the cost function, you are trying to learn two types of variables, U and P, which are represented as $U \times P^T$. The actual cost of the function is the following sum:

$$\| R - U \times P^T \|_2 = \sum_{i,j} \left(R_{i,j} - u_i \times p_j \right)$$

Plus the regularization term. Since Us and Vs are unknown variables, it makes these cost functions non-convex. If you fix P and optimize for U, you essentially can reduce the problem to linear regression. This is a massive win, and the process can be parallelized. ALS is a two-step iterative process. In each iteration, P is fixed and U is solved. In the subsequent step, U is fixed and P is solved. Since ordinary least squares guarantee the minimal mean squared error, the cost function can never increase. Alternating between the steps will reduce the cost function until convergence.

Collaborative filtering is great for recommendations when the least amount of data is available. But you have to be aware of its limitations. The latent features that you generate as part of matrix decomposition are not interpretable. You can also run into the issue of a cold start. When a new item doesn't have a sizable number of users, the model will fail in making a recommendation for the new item and can fall prey to popularity bias.

We now demonstrate the implementation of collaborative filtering by using a made-up dataset containing three columns: `userid`, `movieid`, and `ratings` of multiple users and movies. All the code is provided in separate Jupyter Notebooks for easy access.

```
# Import Sparksession
from pyspark.sql import SparkSession
spark=SparkSession.builder.appName("CF").getOrCreate()
# Print PySpark and Python versions
import sys
print('Python version: '+sys.version)
print('Spark version: '+spark.version)
# Read data
file_location = "cf_data.csv"
file_type = "csv"
infer_schema = "false"
first_row_is_header = "true"
df = spark.read.format(file_type)\
.option("inferSchema", infer_schema)\
.option("header", first_row_is_header)\
.load(file_location)
# Print metadata
df.printSchema()
#  Count data
df.count()
print('The total number of records in the credit card dataset are '+str(df.
count()))
# Import appropriate libraries
from pyspark.sql.types import *
import pyspark.sql.functions as sql_fun
from pyspark.ml.recommendation import ALS, ALSModel
from pyspark.mllib.evaluation import RegressionMetrics, RankingMetrics
from pyspark.ml.evaluation import RegressionEvaluator
import re
# Casting variables
int_vars=['userId','movieId']
for column in int_vars:
```

```
      df=df.withColumn(column,df[column].cast(IntegerType()))
float_vars=['rating']
for column in float_vars:
      df=df.withColumn(column,df[column].cast(FloatType()))
(training, test) = df.randomSplit([0.8, 0.2])
```

We use ALS to perform matrix factorization in this code. Parameters of the ALS model in PySpark are as follows:

- `NumBlocks` is the number of blocks the users and items will be partitioned into in order to parallelize computation.
- `rank` is the number of latent features in the model.
- `maxIter` is the maximum number of iterations to run.
- `regParam` specifies the regularization parameter in ALS to prevent overfitting by penalizing large coefficients.
- `implicitPrefs` specifies whether to use the explicit feedback ALS variant or one adapted for implicit feedback data (defaults to false, which means using explicit feedback).
- `alpha` is a parameter applicable to the implicit feedback variant of ALS that governs the baseline confidence in preference observations (defaults to 1.0).

We set `coldStartStrategy` to `drop` to eliminate empty evaluation metrics.

```
# Define ALS model parameters and fit the model
als = ALS(rank=15,maxIter=2, regParam=0.01,
          userCol="userId", itemCol="movieId", ratingCol="rating",
          coldStartStrategy="drop",
          implicitPrefs=False)
model = als.fit(training)
# Evaluate the model by computing the RMSE on the test data
predictions = model.transform(test)
evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating",
                                predictionCol="prediction")
rmse = evaluator.evaluate(predictions)
print("Root-mean-square error = " + str(rmse))
```

```
# Generate top 10 movie recommendations for each user
userRecs = model.recommendForAllUsers(10)
userRecs.count()
# Generate top 10 user recommendations for each movie
movieRecs = model.recommendForAllItems(10)
movieRecs.count()
userRecs_df = userRecs.toPandas()
print(userRecs_df.shape)
movieRecs_df = movieRecs.toPandas()
print(movieRecs_df.shape)
userRecs_df.head()
```

Output:

	userId	recommendations
0	190174	[(1338, 9.24465560913086), (72028, 9.121529579...
1	190227	[(3812, 7.743837356567383), (2662, 7.072795867...
2	190387	[(3943, 9.847469329833984), (2092, 9.769483566...
3	190540	[(361, 9.0469970703125), (2489, 8.711124420166...
4	190348	[(3912, 8.182646751403809), (995, 8.0464324951...

EXERCISE 7-2: COLLABORATIVE FILTERING

Question 1: Change the number of latent features in the ALS settings and observe how the mean squared error changes.

Summary

- You got an overview of unsupervised learning techniques.
- You now know the differences between hierarchical and non-hierarchical clustering methods, and you know how Silhouette coefficients help in determining the optimal number of clusters.

- You explored how latent Dirichlet allocation works and implemented it.
- You also reviewed the implementation of collaborative filtering using alternating least squares.

Great job! You are now familiar with some of the key concepts that will be useful in unsupervised learning and unstructured data. We take you through a journey of how you can structure your projects and create automated machine learning flows in the next chapter.

CHAPTER 8

Machine Learning Flow and Automated Pipelines

Putting a model into production is one of the most challenging tasks in the data science world. It is one of those last-mile problems that persists in many organizations. Although there are many tools for managing workflows, as the organization matures its needs change, and managing existing models can become a Herculean task. When you take a step back and analyze why it is so challenging, you can see that it is because of the structure that exists in most organizations. An engineering team maintains the production platform. There is a gap between the data science toolset and the production platforms. Some data science work can be developed in a Jupyter Notebook, with little consideration given to the cloud environment. Some of the data flows are created locally with limited scaling. Such applications tend to falter with large amounts of data. Best practices in the software development cycle don't work well with the machine learning lifecycle because of the variety of tasks involved. The standard is mainly defined by the data science team in the organization. Also, rapid developments in the field are leaving vacuums with respect to the management and deployment of models.

This chapter aims to explore the tools available for managing and deploying data science pipelines. We walk you through some of the tools available in the industry and demonstrate with examples how to construct an automated pipeline. We use all the concepts you have learned to construct these pipelines. By the end of this chapter, you will learn how to build machine learning blocks and automate them in coherence with custom data preprocessing.

© Ramcharan Kakarla, Sundar Krishnan, Balaji Dhamodharan and Venkata Gunnu 2024
R. Kakarla et al., *Applied Data Science Using PySpark*, https://doi.org/10.1007/979-8-8688-0820-3_8

This chapter covers the following topics:

- **MLflow.** We introduce MLflow and its components and advantages, followed by a demonstration. In the first half, we focus on the installation and introduction of concepts. In the second half, we focus on implementing a model in MLflow.
- **Automated machine learning pipelines.** We introduce the concept of designing and implementing automated ML frameworks in PySpark. The first half of the section is heavily focused on implementation, and the second half focuses on the outputs generated from the pipeline.

MLflow

With the knowledge obtained from previous chapters, you know how to build individual supervised and unsupervised models. Often, you iterate through multiple models to select the best one. How do you keep track of all the experiments and the hyperparameters you used? How can you reproduce a particular result after multiple experiments? How do you move a model into production when multiple deployment tools and environments exist? Is there a better way to manage the machine learning workflow and share work with the wider community? These are some questions you will have after you become comfortable building models in PySpark. Databricks has built a tool named *MLflow* that can gracefully handle some of the preceding questions while managing the machine learning workflow. MLflow is designed to be open interface and open-source. People often go back and forth between experimentation and adding new features to the model. MLflow handles such experimentation with ease. It is built around REST APIs that multiple tools can consume easily. This framework also makes it easy to add existing machine learning blocks of code. Its open-source nature makes it easy to share the workflows and models across different teams. In short, MLflow offers a framework to manage the machine learning lifecycle, including experimentation, reproducibility, deployment, and central model registry.

MLflow's components are shown in Figure 8-1 and described in the following list.

Figure 8-1. *MLflow's components*

- **Tracking:** This component aids in recording, logging code, configurations, data, and results. It also provides a mechanism for query experiments.
- **Projects:** This component helps bundle the machine learning code in a format that allows easy replication on multiple platforms.
- **Models:** This component assists in deploying machine learning code in diverse environments.
- **Model Registry:** This component acts as a repository to store, manage, and search models.

This framework opens the tunnel for data engineers and data scientists to collaborate and efficiently build data pipelines. It also provides the ability to consume and write data from and to many different systems. It opens up a common framework for working with both structured and unstructured data in data lakes and on batch streaming platforms.

This may seem abstract to first-time users. We walk you through an example of how these components can be useful and implemented. For illustration purposes, we use the same bank dataset used in Chapter 6.

MLflow Code Setup and Installation

In this section, we walk through the code changes necessary to support MLflow, along with its installation. We use a PySpark Docker version for all demonstrations. There are two main steps to get MLflow up and running, as follows:

- Code/script changes
- Docker side changes and MLflow server installation

The code changes are minimal. You must add a few MLflow functions to existing machine learning code to accommodate MLflow.

Regular code works well for building a simple random forest model, but what if you want to know the Receiver Operating Characteristic (ROC)/other metrics (accuracy, misclassification, etc.) by changing the input variables or any hyperparameter settings? Well, if it's a single change, you can record each run in Excel, but this is not an efficient way of tracking your experiments. As data science professionals, we run hundreds of experiments, especially in the model fine-tuning phase. Is there a better way to capture and annotate these results and logs? Yes, you can do this via the MLflow tracking feature. Next we rewrite a simple random forest model code, adding MLflow components. Changes in the code are highlighted in bold.

Note We will save the following code as `.py` and run it via Docker after installing MLflow.

```
# Import libraries
import pyspark
from pyspark.sql import SparkSession
import mlflow
import mlflow.spark
import sys
import time
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.evaluation import BinaryClassificationEvaluator
```

Observe that we have imported the libraries related to MLflow that allow the user to use PySpark. MLflow supports various languages and packages, including Spark and Python, with extended support for packages like `scikitlearn` and `tensorflow`.

```
spark = SparkSession.builder.appName("mlflow_example").getOrCreate()
# show_versions(spark)

filename = "bank-full.csv"
target_variable_name = "y"
from pyspark.sql import functions as F
df = spark.read.csv(filename, header=True, inferSchema=True, sep=';')
```

```
df = df.withColumn('label', F.when(F.col("y") == 'yes', 1).otherwise(0))
df = df.drop('y')
train, test = df.randomSplit([0.7, 0.3], seed=12345)

f or k, v in df.dtypes:
    if v not in ['string']:
        print(k)

df = df.select(['age', 'balance', 'day', 'duration', 'campaign', 'pdays',
'previous', 'label'])

#assemble individual columns to one column - 'features'
def assemble_vectors(df, features_list, target_variable_name):
    stages = []
    #assemble vectors
    assembler = VectorAssembler(inputCols=features_list,
    outputCol='features')
    stages = [assembler]
    #select all the columns + target + newly created 'features' column
    selectedCols = [target_variable_name, 'features']
    #use pipeline to process sequentially
    pipeline = Pipeline(stages=stages)
    #assembler model
    assembleModel = pipeline.fit(df)
    #apply assembler model on data
    df = assembleModel.transform(df).select(selectedCols)
    return df

#exclude target variable and select all other feature vectors
features_list = df.columns
#features_list = char_vars #this option is used only for ChiSqselector
features_list.remove('label')

# apply the function on our DataFrame
assembled_train_df = assemble_vectors(train, features_list, 'label')
assembled_test_df = assemble_vectors(test, features_list, 'label')
```

```
print(f"sysarg1: {sys.argv[1]}")
print(f"sysarg2: {sys.argv[2]}")
print(f"sysarg3: {sys.argv[3]}")
```

These are the system arguments you intend to change in each experiment. You will pass these as system arguments when you execute this script. Here, we used `maxBins` and `maxDepth`. You can include any other variable changes you want to log across the script.

```
maxBinsVal = float(sys.argv[2])
maxDepthVal = float(sys.argv[4])
```

In the following step, we are initializing the machine learning code part with MLflow since the tracking variables of interest are within this scope. You can modify the scope as intended depending on the use case.

```
# Set our tracking server uri for logging
mlflow.set_tracking_uri(uri="http://127.0.0.1:8080")

# Create a new MLflow Experiment
mlflow.set_experiment("MLflow PySpark Example")

with mlflow.start_run():
    stages_tree=[]

#Define stages for pipeline
classifier = RandomForestClassifier(labelCol = 'label',featuresCol =
'features',maxBins=maxBinsVal, maxDepth=maxDepthVal)
    stages_tree += [classifier]
    pipeline_tree=Pipeline(stages=stages_tree)
    print('Running RFModel')
    RFmodel = pipeline_tree.fit(assembled_train_df)
    print('Completed training RFModel')
    predictions = RFmodel.transform(assembled_test_df)
    evaluator = BinaryClassificationEvaluator()

print("Test Area Under ROC: " + str(evaluator.evaluate(predictions,
{evaluator.metricName: "areaUnderROC"})))
# Log parameters and metrics to MLflow
    mlflow.log_param("maxBins", maxBinsVal)
    mlflow.log_param("maxDepth", maxDepthVal)
```

```
mlflow.log_metric("ROC", evaluator.evaluate(predictions, {evaluator.
metricName: "areaUnderROC"}))
    mlflow.spark.log_model(RFmodel,"spark-model")

# Set a tag that we can use to remind ourselves what this run was for
    mlflow.set_tag("Training Pipeline", "Basic Sklearn model for bank-
    full.csv")
# Log the model with MLflow
    model_info = mlflow.spark.log_model(RFmodel,
        artifact_path="sklearn-model",
        registered_model_name="sklearn-quickstart",)
```

In the preceding code snippet, we added `log_param` and `log_metric` to capture the information we want to keep tabs on. Also, note that we are logging the model using the `mlflow.spark.log_model` function, which helps in saving the model on the MLflow backend. This is an optional statement, but it is handy to register the model from MLflow. You can accommodate an existing model with MLflow components with minimal code changes. Why is this important? Tracking the change in parameters and metrics can become challenging with hundreds of experiments staged over multiple days.

We now need to save the preceding code as a Python file. We use a Spark submit to execute this PySpark code. In this illustration, we saved the code as `Chapter8_mlflow_example.py`.

Moving on to the second part of MLflow setup, is a simple three-step process, as mentioned:

1. `pip install mlflow`
2. Initiate the MLflow server:

   ```
   mlflow server --host 127.0.0.1 --port 8080
   ```

3. Open a browser and enter `http://127.0.0.1:8080/` to open the MLflow UI

MLflow User Interface Demonstration

In this section, we walk through the MLflow user interface with an example discussed in the previous section. The following code runs the script saved earlier with three different settings:

```
spark-submit --master local chapter8_mlflow_example.py --maxBinsVal
16 --maxDepthVal 3
spark-submit --master local chapter8_mlflow_example.py --maxBinsVal
16 --maxDepthVal 5
spark-submit --master local chapter8_mlflow_example.py --maxBinsVal
32 --maxDepthVal 5
```

Once we finish running the preceding three commands, we can navigate back to the MLflow user interface. (On any web browser, navigate to `http://127.0.0.1:8080`.) The MLflow user interface is shown in Figure 8-2.

As you can see, the user interface has all the information formatted into experiments and models. You can make any annotations or notes using the notes options about each of the runs in each experiment. This framework allows you to manage and run multiple experiments with multiple runs. You can also filter the runs based on parameter settings. This is extremely useful in a machine learning lifecycle because, unlike the software development lifecycle, we tend to iterate between older and newer versions based on the stability and accuracy of the models.

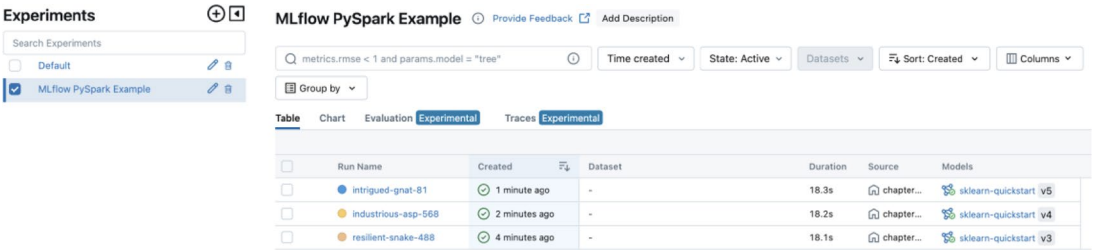


Figure 8-2. *MLflow UI*

There is also detailed row-level information on each run. You can drill down and compare this information across many experiments. For the illustration here, we have used only a single metric, but in a real-world scenario, you may also want to capture accuracy, misclassification, lift, and KS statistics for each run. You could then compare and sort each iteration based on a metric that is acceptable based on business needs.

When you select a couple of iterations from the prior executions, observe that the Compare button is active; it is grayed out in Figure 8-2. When you click Compare, it will compare these two iterations and give prepared graphs on how metrics vary *across different runs* (see Figure 8-3). In the preceding code, we explicitly set the log metrics. From version 3 of Spark, there is an option for automatic logging.

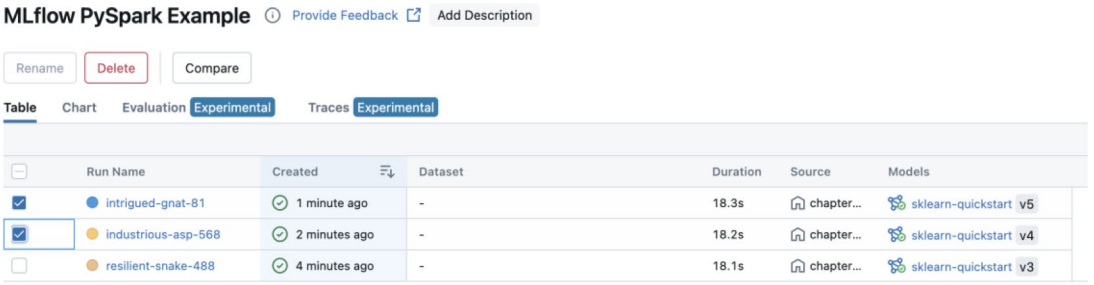


Figure 8-3. *MLflow UI comparison of runs*

Observe the window in Figure 8-4; MLflow autogenerates the RunID. This feature allows you to compare multiple runs in an experiment. You can visually pick the model runs that perform best when you have enough data points. You can also observe how the model performance can vary by changing the parameters.

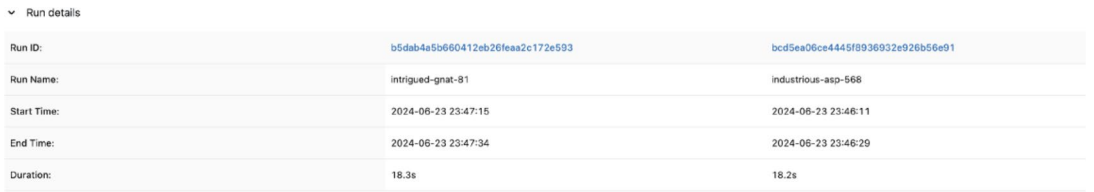


Figure 8-4. *MLflow UI comparison window*

If you go back to the homepage in the MLflow UI and click any run, it will give you more information about the model run, as shown in Figure 8-5. This Run window also provides the handy feature of registering the model. Registering a model will make it available for scoring. Multiple other features will open upon registering the model.

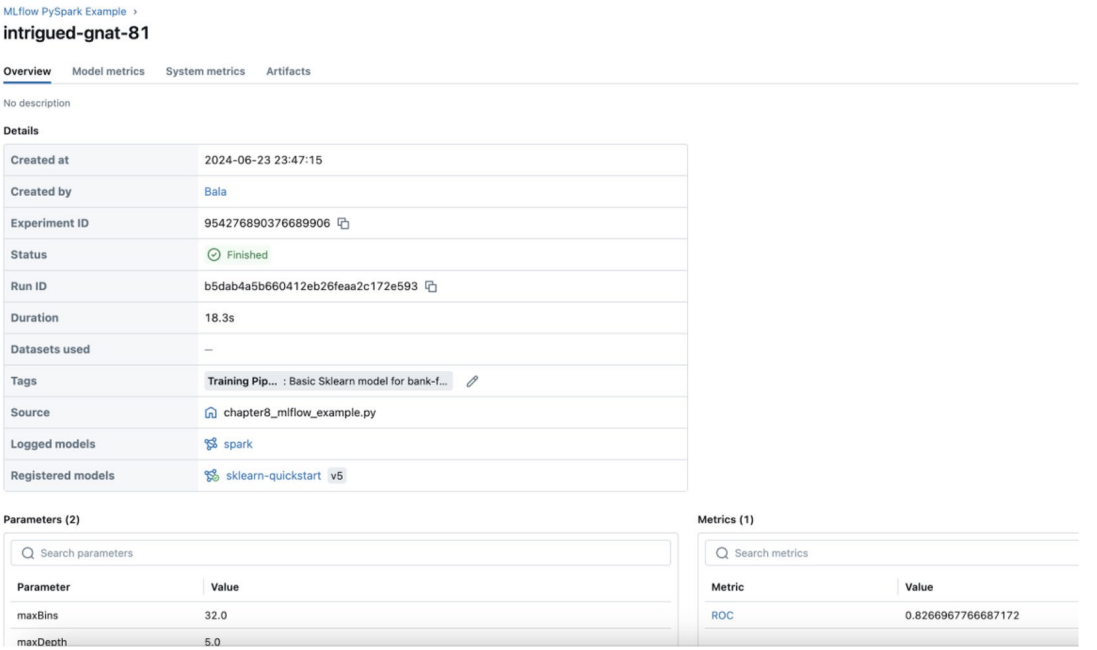


Figure 8-5. *MLflow UI Run window*

Clicking the `spark-model` file in the Artifacts section will open the next window. This window contains all the metadata information of the model required for running it, as shown in Figure 8-6.

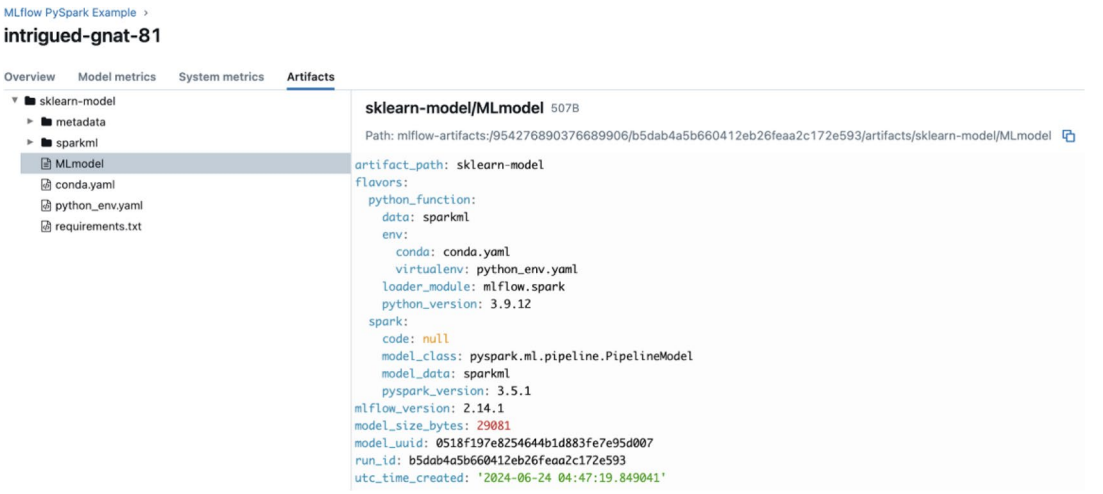


Figure 8-6. *MLflow UI model information*

The user can also register the model manually. However, we already registered the model via code. The registered models can be viewed by clicking the `Model Registered` link in the top-right corner. In the Models tab, the models are automatically version controlled.

Figure 8-7. *MLflow Models tab*

Note Figure 8-7 shows Version 5 since the model registration code was run a few times. In your case, you may notice Version 1.

Well, all this is great, but how can you score new data from these pipelines? When you click the run or the model information, you get the full path of the model metadata information, represented in Figure 8-7. You can use this metadata to score any new data. There are multiple ways this can be achieved. As an example, we write some simple code to prepare the data for scoring, and we then use the `best_roc_model`. We save the following code as a Python file and run it via `spark-submit` in Docker.

```
#Importing necessary libraries
import mlflow
import mlflow.spark
import sys
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("mlflow_predict").getOrCreate()
filename = "bank-full.csv"
target_variable_name = "y"
from pyspark.sql import functions as F
df = spark.read.csv(filename, header=True, inferSchema=True, sep=';')
df = df.withColumn('label', F.when(F.col("y") == 'yes', 1).otherwise(0))
df = df.drop('y')
train, test = df.randomSplit([0.7, 0.3], seed=12345)
for k, v in df.dtypes:
```

```
    if v not in ['string']:
        print(k)
df = df.select(['age', 'balance', 'day', 'duration', 'campaign', 'pdays',
'previous', 'label'])
from pyspark.ml.feature import VectorAssembler
from pyspark.ml import Pipeline
# assemble individual columns to one column - 'features'
def assemble_vectors(df, features_list, target_variable_name):
    stages = []
    #assemble vectors

assembler = VectorAssembler(inputCols=features_list, outputCol='features')
    stages = [assembler]
    #select all the columns + target + newly created 'features' column
    selectedCols = [target_variable_name, 'features']
    #use pipeline to process sequentially
    pipeline = Pipeline(stages=stages)
    #assembler model
    assembleModel = pipeline.fit(df)
    #apply assembler model on data
    df = assembleModel.transform(df).select(selectedCols)
    return df
#exclude target variable and select all other feature vectors
features_list = df.columns
#features_list = char_vars #this option is used only for ChiSqselector
features_list.remove('label')
# apply the function on our dataframe
assembled_test_df = assemble_vectors(test, features_list, 'label')
print(sys.argv[1])
# model information from argument
model_uri=sys.argv[1]
print("model_uri:", model_uri)
# Load the Spark ML model from MLflow
model = mlflow.spark.load_model(model_uri)
print("model.type:", type(model))
predictions = model.transform(assembled_test_df)
```

```
print("predictions.type:", type(predictions))
predictions.printSchema()
df = predictions.select('rawPrediction','probability', 'label', 'features')
df.show(5, False)
```

This code is a simple pipeline. You also have the flexibility of wrapping the scoring function as a simple `udf` as well. After saving the code, you need to run the following command in Docker for results:

```
spark-submit --master local chapter8_predict_spark.py
mlartifacts/795206614136056686/cd6d4c9a47b54d1396f63eb172fbc547/artifacts/
sklearn-model
```

This will give you the output shown in Figure 8-8.

```
+-----------------------------------------+-----------------------------------------+-----+-----------------------------------+
|rawPrediction                            |probability                              |label|features                           |
+-----------------------------------------+-----------------------------------------+-----+-----------------------------------+
|[14.682279961762092,5.317720038237908]   |[0.7341139980881046,0.26588600191189543]|0    |[18.0,35.0,21.0,104.0,2.0,-1.0,0.0]|
|[16.794318838126294,3.2056811618737067]  |[0.8397159419063147,0.16028405809368534]|0    |[19.0,56.0,12.0,246.0,1.0,-1.0,0.0]|
|[13.322721722519985,6.677278277480018]   |[0.6661360861259992,0.33386391387400083]|1    |[19.0,134.0,27.0,271.0,2.0,-1.0,0.0]|
|[18.483022065155744,1.5169779348442576]  |[0.9241511032577872,0.07584889674221287]|0    |[19.0,96.0,23.0,106.0,3.0,-1.0,0.0]|
|[14.427935035462232,5.572064964537771]   |[0.7213967517731115,0.2786032482268885] |0    |[19.0,291.0,5.0,291.0,5.0,-1.0,0.0]|
+-----------------------------------------+-----------------------------------------+-----+-----------------------------------+
only showing top 5 rows
```

Figure 8-8. *MLflow model scoring output*

All the preceding content and code are specified in Spark. MLflow supports multiple other flavors. You can serialize the preceding pipeline in the `mleap` flavor, a project to host Spark pipelines without a Spark context for smaller datasets where you don't need distributed computing. MLflow can also publish code to GitHub in the MLflow project format, making it easy for anyone to run the code.

Automated Machine Learning Pipelines

The machine learning lifecycle is an iterative process. You'll often go back and forth tuning parameters, inputs, and data pipelines, which can quickly become cumbersome with the number of data management pipelines. Creating automated pipelines can save a significant amount of time. You have learned data manipulations, algorithms, and modeling techniques from the earlier chapters. Now, you'll combine them to create an automated PySpark flow that can generate baseline models for quick experimentation.

This experiment uses a churn dataset from Kaggle (https://www.kaggle.com/shrutimechlearn/churn-modelling#Churn_Modelling.csv). This banking dataset contains data about attributes of customers and who has churned. The objective is to identify customers who will churn based on given attributes. Table 8-1 lists the attributes.

Table 8-1. *Metadata*

Column	Description
RowNumber	Identifier
CustomerId	Unique ID for the customer
Surname	Customer's last name
Credit score	Credit score of the customer
Geography	The country to which the customer belongs
Gender	Male or female
Age	Age of customer
Tenure	Number of years the customer has been with the bank
Balance	Bank balance of the customer
NumOfProducts	Number of bank products the customer is utilizing
HasCrCard	Binary flag for whether the customer holds a credit card with the bank or not
IsActiveMember	Binary flag for whether the customer is an active member with the bank or not
EstimatedSalary	Estimated salary of the customer in dollars
Exited	Binary flag: 1 If the customer closed their account and 0 if the customer still has an account

Pipeline Requirements and Framework

When designing any pipeline, it is imperative to define the outputs and requirements you expect. First, you define the outputs that you think are necessary. Since this example tackles a binary target, it would be nice to have the following outputs:

- **KS:** The Kolmogorov-Smirnov test is a measure of separation between the goods and the bads. Higher is better.
- **ROC:** This curve is a way to compare diagnostic tests. It is a plot of the true positive rate against the false positive rate. Higher is better.
- **Accuracy:** This is true positives+true negatives)/true positives+true negatives+false positives+false negatives. Higher is better.

Here are the requirements for an automated model-building tool.

- The module should be able to handle the missing values and categorical values by itself.
- The module should take care of variable selection.
- This module should test multiple algorithms before selecting the final variables.
- The module should compare different algorithms based on preferred metrics and select the champion and challenger models.
- It would be handy if the module could remove certain variables by prefix or suffix or by variable names. This can power multiple iterations and tweak input variables.
- The module should be able to collect and store all the output metrics and collate them to generate documents for later reference.
- Finally, it would be nice if the module could save the model objects and autogenerate the scoring codes for seamless deployment.

How do we determine these requirements? Remember that we discussed the CRISP–DM framework in Chapter 3 (see Figure 8-9)? We are using this framework here as the architecture blueprint.

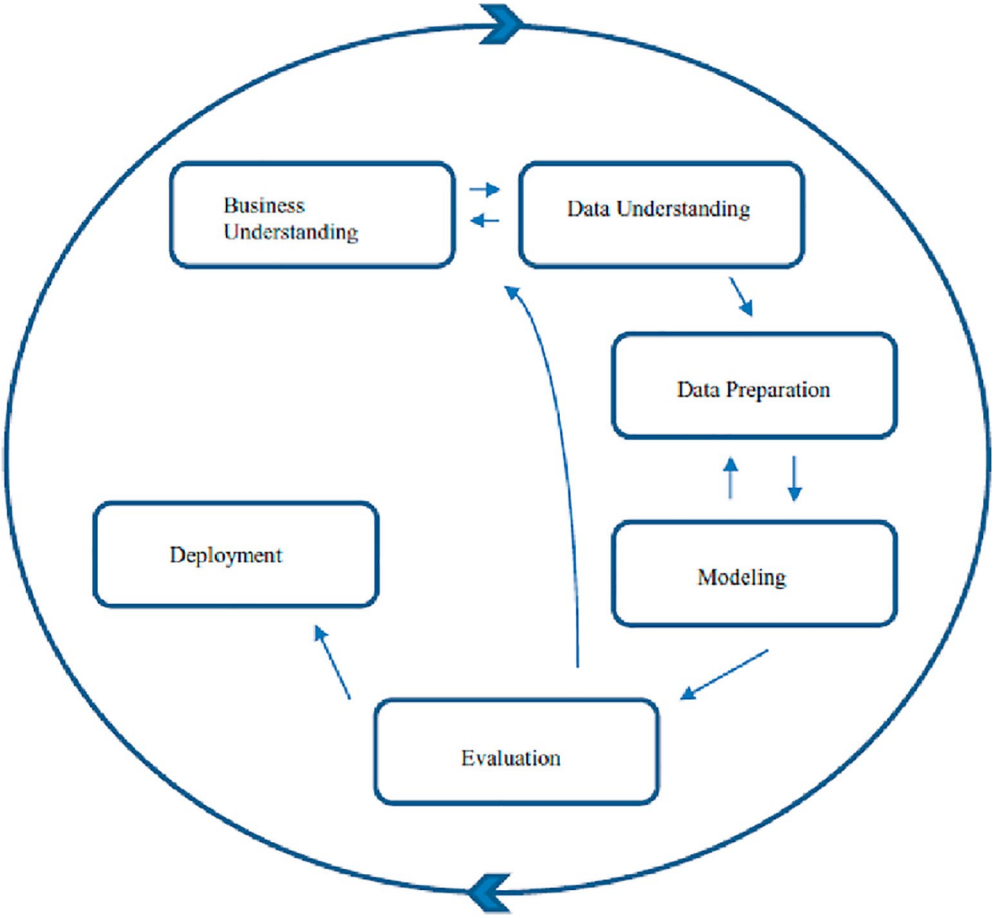


Figure 8-9. *CRISP–DM*

These are a lot of requirements and can quickly become overwhelming if not handled properly. We have defined the outputs and requirements, but what will the expected inputs be? Probably a dataset either from a datastore or a flat file. With this information, you can start building the pipelines by breaking the code into logical steps. This will be a code-intensive chapter, and you can use this module and create your own baseline automated models. We break tasks into subtasks in each of the logical steps. These are broadly divided into the following:

- Data manipulations
- Feature selection
- Model building

- Metrics calculation
- Validation and plot generation
- Model selection
- Score code creation
- Collating results
- Framework to handle all the preceding steps

Data Manipulations

In this section, we define some commonly used functions for handling the data, such as the following:

- Missing value percentage calculations
- Metadata categorization of input data
- Handling categorical data using label encoders
- Imputing missing values
- Renaming categorical columns
- Combining features and labels
- Data splitting to training, testing, and validation
- Assembling vectors
- Scaling input variables

You can access this code from this link: https://github.com/Apress/applied-data-science-using-pyspark/blob/main/Ch08/Chapter8_automator/data_manipulations.py.

Feature Selection

This module shortlists the variables and selects the top variables that are most predictive. It uses random forests to identify the top variables. You can access the code for this module from this link: https://github.com/Apress/applied-data-science-using-pyspark/blob/main/Ch08/Chapter8_automator/feature_selection.py.

Model Building

In this module, we define multiple algorithm functions that support binary targets, including logistic regression, random forests, gradient boosting, and neural nets. You can access the code for this module from this link: https://github.com/Apress/applied-data-science-using-pyspark/blob/main/Ch08/Chapter8_automator/model_builder.py.

Metrics Calculation

This module calculates the model metrics, including KS, ROC, and accuracy. You can access the code for this module from this link: https://github.com/Apress/applied-data-science-using-pyspark/blob/main/Ch08/Chapter8_automator/metrics_calculator.py.

Validation and Plot Generation

This module generates the various plots, including ROC, confusion matrix, and KS. Model validation is also done as part of this module. You can access the code at https://github.com/Apress/applied-data-science-using-pyspark/blob/main/Ch08/Chapter8_automator/validation_and_plots.py.

Model Selection

This module generates the challenger and champion model definitions based on validation metrics criteria. It also generates the consolidated Excel output file containing all the models' information. The code is provided at https://github.com/Apress/applied-data-science-using-pyspark/blob/main/Ch08/Chapter8_automator/model_selection.py.

Score Code Creation

This module generates the pseudo-score code for production deployment. It links all the model objects generated in the modeling process and places them in a single location. This also includes data manipulation and categorical variable processing functions. This

file can be run independently as long as the model objects are saved in the same location and are pointing to the right input data source. The code for this module is at https://github.com/Apress/applied-data-science-using-pyspark/blob/main/Ch08/Chapter8_automator/scorecode_creator.py.

Collating Results

A zip file module is used to bind all the output generated by the earlier modules into a single ZIP file. This file can be easily exported to deploy models elsewhere. The link for this module is https://github.com/Apress/applied-data-science-using-pyspark/blob/main/Ch08/Chapter8_automator/zipper_function.py.

Framework

This framework file combines all the preceding modules to orchestrate them logically and create the desired output. It treats the data and performs variable selection. It builds machine learning algorithms and validates the models on holdout datasets. It picks the best algorithm based on user-selected statistics. It also produces the scoring code for production.

This framework file receives all the inputs. We save all the preceding modules, including this framework file, as a Python file. Since we are dealing with a churn modeling dataset here, we provide the input CSV file location in the `data_folder_path`. All the other required inputs are provided in the first block of the code. This code is designed to accommodate the Docker version of PySpark. By making minimal changes, it can also be adapted to any cluster execution. You can access the code for this at https://github.com/Apress/applied-data-science-using-pyspark/blob/main/Ch08/Chapter8_automator/build_and_execute_pipe.ipynb or https://github.com/Apress/applied-data-science-using-pyspark/blob/main/Ch08/Chapter8_automator/build_and_execute_pipe.py.

For execution, open Docker using the following command:

```
docker run -it -p 8888:8888 -v /Users/ramcharankakarla/demo_data/:/home/
jovyan/work/ jupyter/pyspark-notebook:latest bash
```

On the local machine, copy all the files to the path exposed to Docker (e.g., `/Users/ramcharankakarla/demo_data/`). We created a folder called `Chapter8_automator` and placed all the required files there (see Table 8-2). Copy the `churn_modeling.csv` dataset to that same folder.

Table 8-2. *Files and Their Functions in the Automation Framework*

Module	Filename
Data manipulations	`data_manipulations.py`
Feature selection	`feature_selection.py`
Model building	`model_builder.py`
Metrics calculation	`metrics_calculator.py`
Validation and plot generation	`validation_and_plots.py`
Model selection	`model_selection.py`
Score code creation	`scorecode_creator.py`
Collating results	`zipper_function.py`
Framework	`build_and_execute_pipe.py or build_and_execute_pipe.ipynb`

Before running these framework files, make sure you install or check for the existence of the following packages in Docker:

```
pip install openpyxl
pip install xlsxwriter
```

To run the framework file from Docker, use the following command:

```
spark-submit --master local[*] /home/jovyan/work/Chapter8_automator/build_and_execute_pipe.py
```

After the execution is completed, all the data will be stored in `/home/jovyan/`.

To make a local copy of the files and metrics and prevent loss of any work, use the following command. This command creates a local copy of the work. Similarly, you can copy a zip file (`mla_chapter8_testrun.zip`) if needed.

```
cp -r mla_chapter8_testrun /home/jovyan/work/
```

Nice job! You have successfully built an end-to-end automation engine that can save you a significant amount of time. The next section looks at the output this engine has generated.

Pipeline Outputs

Figures 8-10 through 8-12 and Tables 8-3 through 8-5 illustrate the outputs generated by the automated pipeline framework. The following outputs are generated for any given binary model.

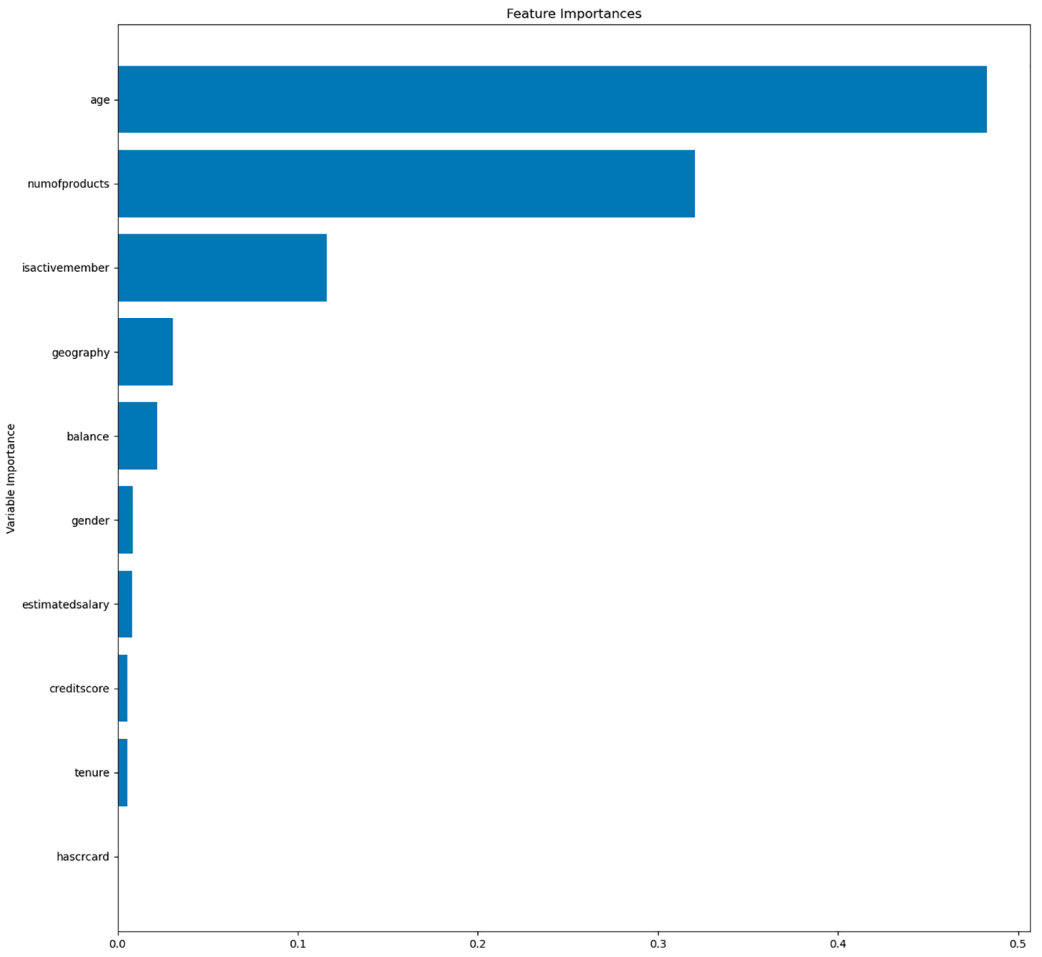


Figure 8-10. *Automated model feature importance*

Table 8-3. *Combined Metrics*

model_type	roc_train	accuracy_ train	ks_ train	roc_valid	accuracy_ valid	ks_ valid
neuralNetwork	0.86171092	0.858369099	55.5	0.857589	0.862000986	54.43
randomForest	0.838162018	0.860228898	50.57	0.835679	0.856086742	50.48
logistic	0.755744261	0.81230329	37.19	0.754326	0.807294234	37.31
gradientBoosting	0.881999528	0.869384835	59.46	0.858617	0.853622474	55.95

model_type	roc_test	accuracy_test	ks_test	roc_oot1	accuracy_oot1
neuralNetwork	0.813792	0.849134	46.58	0.856175	0.8582
randomForest	0.806239	0.840979	44.3	0.834677	0.8575
logistic	0.719284	0.805301	32.19	0.751941	0.8106
gradientBoosting	0.816005	0.850153	47.22	0.870804	0.8643

model_type	ks_oot1	roc_oot2	accuracy_oot2	ks_oot2	selected_model
neuralNetwork	54.43	0.856175108	0.8582	54.43	Champion
randomForest	49.99	0.834676561	0.8575	49.99	Challenger
logistic	36.49	0.751941185	0.8106	36.49	
gradientBoosting	57.51	0.870803793	0.8643	57.51	

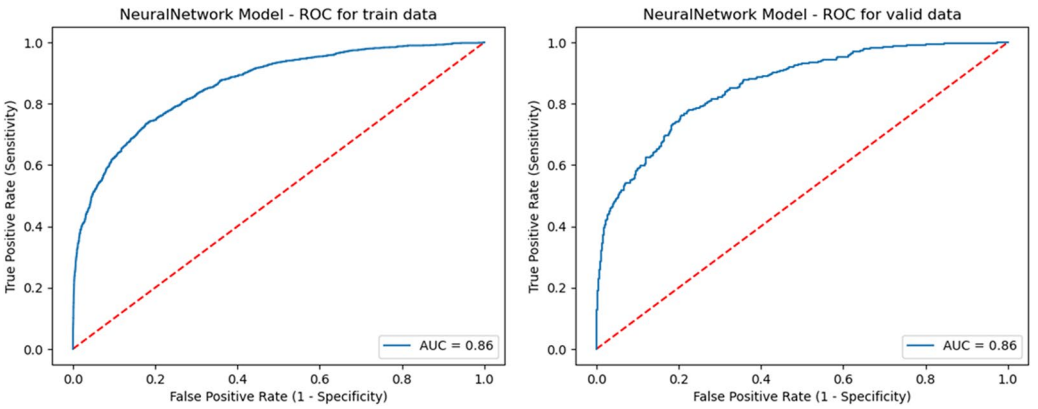

NeuralNetwork Model - ROC for train data
True Positive Rate (Sensitivity)
False Positive Rate (1 - Specificity)
AUC = 0.86
NeuralNetwork Model - ROC for valid data
True Positive Rate (Sensitivity)
False Positive Rate (1 - Specificity)
AUC = 0.86

Figure 8-11. *Automated model ROCs*

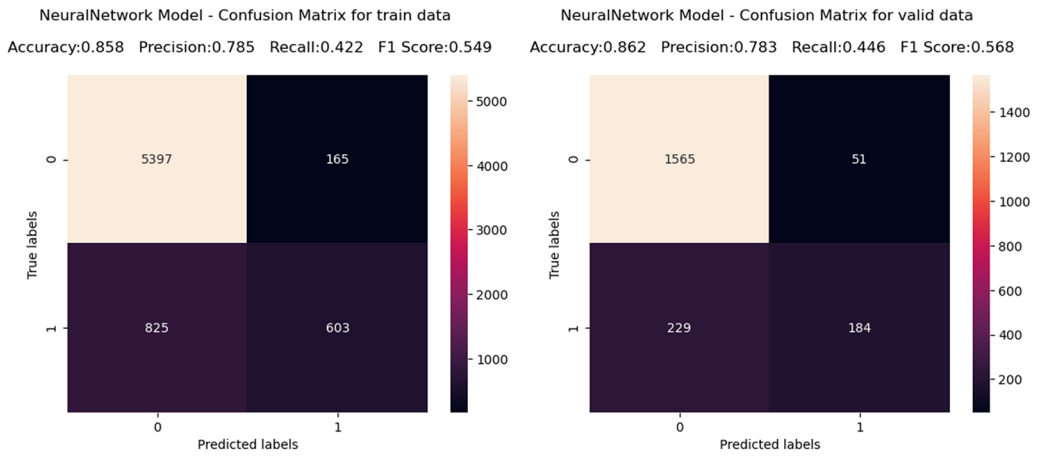


Figure 8-12. *Automated model — confusion matrix*

Table 8-4. *Kolmogorov-Smirnov Statistic*

deciles	count	target	non_target	Pct_target	cum_target	m_non_targ	%Dist_Targe	ist_non_Tar	spread
1	699	577	122	82.54649	577	122	40.40616	2.193456	38.21271
2	699	299	400	42.77539	876	522	61.34454	9.385113	51.95942
3	699	183	516	26.18026	1059	1038	74.15966	18.66235	55.49731
4	699	113	586	16.16595	1172	1624	82.07283	29.19813	52.8747
5	699	99	600	14.16309	1271	2224	89.0056	39.98562	49.01999
6	699	67	632	9.585122	1338	2856	93.69748	51.34844	42.34904
7	699	32	667	4.577969	1370	3523	95.93838	63.34052	32.59785
8	699	33	666	4.72103	1403	4189	98.2493	75.31464	22.93466
9	699	12	687	1.716738	1415	4876	99.08964	87.66631	11.42333
10	699	13	686	1.8598	1428	5562	100	100	0

Table 8-5. *Neural Network Test*

deciles	count	target	non_target	Pct_target	cum_target	m_non_targ	%Dist_Targe	ist_non_Tar	spread
1	99	73	26	73.73737	73	26	37.2449	3.312102	33.9328
2	98	36	62	36.73469	109	88	55.61224	11.21019	44.40205
3	98	23	75	23.46939	132	163	67.34694	20.76433	46.58261
4	98	13	85	13.26531	145	248	73.97959	31.59236	42.38724
5	98	15	83	15.30612	160	331	81.63265	42.16561	39.46705
6	98	17	81	17.34694	177	412	90.30612	52.48408	37.82205
7	98	11	87	11.22449	188	499	95.91837	63.56688	32.35149
8	98	3	95	3.061224	191	594	97.44898	75.66879	21.78019
9	98	4	94	4.081633	195	688	99.4898	87.64331	11.84648
10	98	1	97	1.020408	196	785	100	100	0

This module is intended to help you create quick, automated experiments and is in no way meant to replace the model-building activity.

EXERCISE 8-1: BUILDING CUSTOM PIPELINES

Question: We challenge you to build a separate pipeline or integrate code flow to accommodate continuous targets.

Summary

- You learned the challenges of model management and deployment.
- You now know how to use MLflow for managing experiments and deploying models.
- You explored how to build custom pipelines for various model-building activities.
- You saw how pipelines can be stacked together to create an automated pipeline.

Great job! You are now familiar with some of the key concepts that will be useful in putting a model into production. This should give you a fair idea of how you want to manage a model lifecycle as you are building your data-preparation pipelines. In the next chapter, we cover some tips, tricks, and interesting topics that can be useful in your day-to-day work.

CHAPTER 9

Deploying Machine Learning Models

This chapter covers the best practices to follow when deploying a machine learning model. We use three methods to demonstrate the production deployment, as follows:

- Model deployment using an HDFS object and pickle files
- Model deployment using Docker
- Real-time scoring API

In the previous chapter, you got a glimpse of the first technique. In this chapter, we cover it in more detail. The code provided is the starter code to build the model and evaluate the results.

Starter Code

You can find the code at this GitHub link: https://github.com/Apress/applied-data-science-using-pyspark/tree/main/Ch09. Refer to the `model deployment.ipynb` file in the GitHub repository to access the starter code. This code is used for the rest of the exercise. If you already have a model, you can skip this section.

Save Model Objects and Create Score Code

So far, you have built a solid model and validated it using multiple metrics. It's time to save the model objects for future scoring and create the score code.

© Ramcharan Kakarla, Sundar Krishnan, Balaji Dhamodharan and Venkata Gunnu 2024
R. Kakarla et al., *Applied Data Science Using PySpark*, https://doi.org/10.1007/979-8-8688-0820-3_9

- Model objects are the PySpark or Python parameters that contain information from the training process. You have to identify all the relevant model objects needed to replicate the results without retraining the model. Once identified, they are saved in a specific location (HDFS or other path) and used later during scoring.
- The score code contains the essential code to run your model objects on the new dataset so as to produce the scores. Remember, the score code is used only for the scoring process. You should not have the model-training scripts in a score code.

Model Objects

In this model example, we need six objects to be saved for future use, as follows:

- `char_labels` – This object is used to apply label encoding on new data.
- `assembleModel` – This object is used to assemble vectors and make the data ready for scoring.
- `clf_model` – This object is the random forest classifier model.
- `features_list` – This object has the list of input features to be used from the new data.
- `char_vars` – These are character variables.
- `num_vars` – These are numeric variables.

In the `Model deployment.ipynb` file, we provide a section titled "Save Model Objects for Future Scoring." Refer to that section for information about saving your model objects.

Score Code

A Python script file is used to perform scoring operations. As mentioned, this script file should not contain model-training scripts. At a bare minimum, a score code should do the following tasks:

1) Import necessary packages
2) Read the new input file to perform scoring
3) Read the model objects saved from the training process
4) Perform necessary transformations using model objects
5) Calculate model scores using the classifier/regression object
6) Output the final scores in a specified location

First, create a `helper.py` script that has all the necessary code to perform scoring. The code to paste in the file is provided in the Score Code section of the `Model deployment.ipynb` file.

After you create the file, you can score new data using the final code in `Model deployment.ipynb`. You can save the scripts in a `run.py` file.

The final scores are available in the `final_scores_df DataFrame`, and this dataset is saved as a CSV file in the final step. You need to make sure that the `run.py` and `helper.py` files exist in the same directory. That's it. You have solid score code that is ready to be deployed in a production environment. We go through the production implementation in the next sections.

Model Deployment Using HDFS Object and Pickle Files

This is a very simple way to deploy a model in production. When you have multiple models to score, you can manually submit each `run.py` script that you created in the previous step, or you can create a scheduler to run the scripts periodically. In either case, the following code will help you accomplish this task:

```
spark-submit run.py
```

As mentioned, the `run.py` and `helper.py` files should be in the same directory for this code to work. You can configure the preceding command to specify `configurations` while submitting the job. The `configurations` option is useful when you deal with large datasets, because you might have to play with executors, cores, and memory options. You can read about the `spark-submit` and `configuration` options at the following links:

https://spark.apache.org/docs/latest/submitting-applications.html

https://spark.apache.org/docs/latest/configuration.html

Model Deployment Using Docker

We now switch gears and deploy a model using Docker. Docker-based deployment is useful in a lot of ways when compared to a traditional scoring method, such as `spark-submit`.

- Portability – You can port your application to any platform and instantly make it work.
- Containers – The entire application is self-contained. All the codes and executables needed to make your application work sit within the container.
- Microservices architecture – Instead of a traditional, monolithic application, you can separate it into microservices. This way, you can modify or scale up/down individual micro-services as and when needed.
- Faster software delivery cycles – Compatible with Continuous Integration and Continuous Development (CI/CD) processes make it easy to deploy applications to production.
- Data scientists/engineers don't have to spend additional time fixing their code/bugs when the Spark backend is altered.
- When compared to virtual machines, Docker consumes less memory, making it more efficient in using system resources.

Let's go ahead and implement this model using Docker. Before you start, you should have the following files in a single directory (see Figure 9-1).

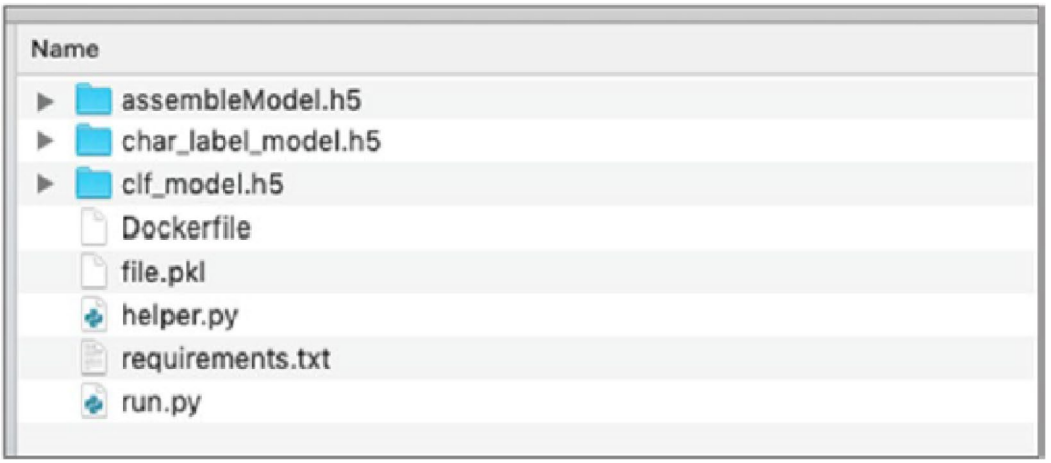


Figure 9-1. *Directory structure*

Most of the files should look familiar to you by now. We have two new files shown in the image: `Dockerfile` and `requirements.txt`. `Dockerfile` contains the scripts to create a Docker container, and `requirements.txt` contains all the additional packages that you need to make your code work. Let's look at the file contents.

The requirements.txt File

The contents of this file are provided here so that you can copy and paste them inside the file (see Figure 9-2).

```
pandas
numpy
matplotlib
seaborn
scikit-learn
requests
flask
py4j
```

Figure 9-2. *The requirements.txt file*

Dockerfile

We are using the base Docker image as the PySpark Jupyter Notebook image. Inside the image, we create a working directory called `/deploy/` into which we will copy and paste all our model objects and files (see Figure 9-3).

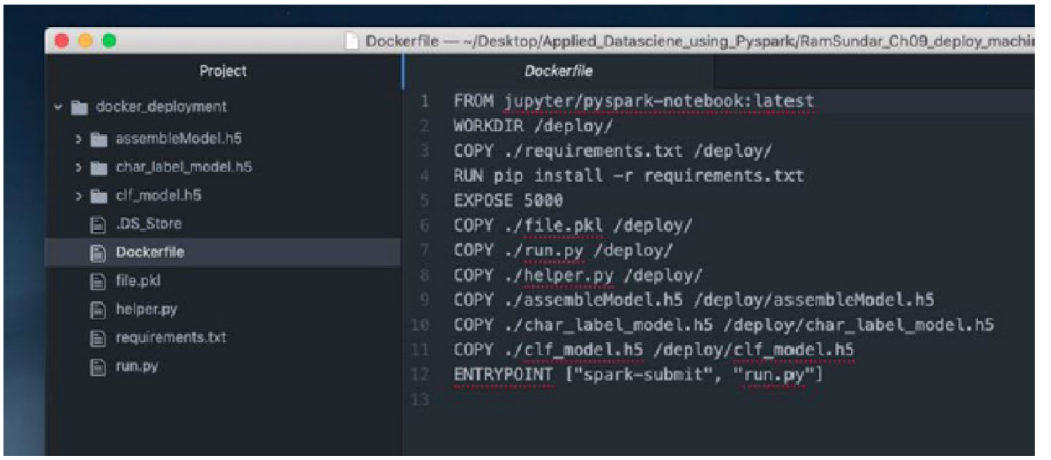


Figure 9-3. *Dockerfile*

This is done in successive steps. In addition, we install the required packages by executing the `requirements.txt` file, which is shown in Line 4. Finally, we execute the model using the `spark-submit` command used in the previous section. One thing to notice is that the port number, 5000, is exposed in the container. We will use this port in the real-time scoring section. Just ignore this port for now. The `Dockerfile` script is provided here:

```
# use  jupyter/pyspark-notebook as the base image
FROM jupyter/pyspark-notebook:latest
WORKDIR /deploy/
# Copy the requirements.txt file from local to container directory
COPY ./requirements.txt /deploy/
# install python dependencies
RUN pip install -r requirements.txt
# Expose port 5000
EXPOSE 5000
# Copy necessary files from local to container dir
```

```
COPY ./file.pkl /deploy/
COPY ./run.py /deploy/
COPY ./helper.py /deploy/
COPY ./assembleModel.h5 /deploy/assembleModel.h5
COPY ./char_label_model.h5 /deploy/char_label_model.h5
COPY ./clf_model.h5 /deploy/clf_model.h5
# Specify the entry point command to run when the container starts
ENTRYPOINT ["spark-submit", "run.py"]
```

Changes Made in the helper.py and run.py Files

In the `helper.py` file, change the following line:

```
path_to_read_objects ='/deploy'
```

In the `run.py` file, change the following lines:

```
path_to_output_scores = '/localuser'
filename = path_to_output_scores + "/score_data.csv"
```

These changes are made to reflect the appropriate directory path within Docker (`/deploy`) and the volume that we will mount later (`/localuser`) during Docker execution.

Create Docker and Execute Score Code

To create the Docker container, execute the following code in the Terminal/command prompt:

```
docker build -t scoring_image .
```

Once the image is built, you can execute it using the following command. You need to execute the command from the directory that contains the scoring data.

```
docker run -p 5000:5000 -v ${PWD}:/localuser scoring_image:latest
```

That's it. You should see the `predictions.csv` file in the local folder. Up to this point, we have used a batch-scoring process. Let's now move toward a real-time scoring API.

Real-Time Scoring API

A real-time scoring API is used to score your models in real-time. It enables your model to be embedded in a web browser in order to get predictions and take action based on those predictions. This is useful for making online recommendations, checking credit card approval status, and so on. The application is enormous, and it becomes easier to manage such APIs using Docker. This example uses Flask to implement the model in real-time (see Figure 9-4). The Flask scripts are stored in the `pysparkapi/app.py` file. You can refer to the code for this section at https://github.com/Apress/applied-data-science-using-pyspark/tree/main/Ch09/real_time_scoring.

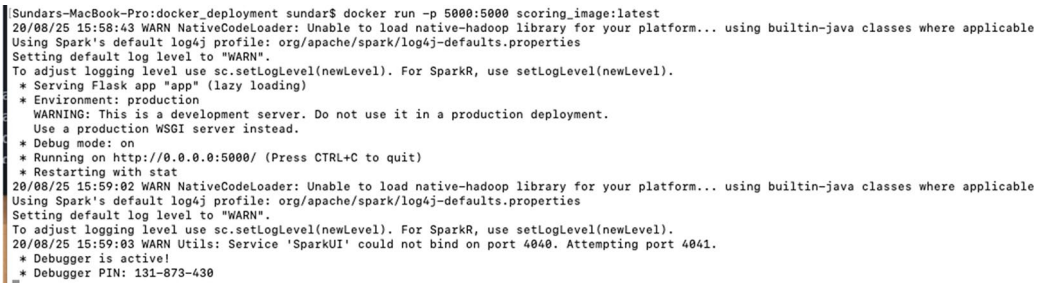

Sundars-MacBook-Pro:docker_deployment sundar$ docker run -p 5000:5000 scoring_image:latest
20/08/25 15:58:43 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
* Serving Flask app "app" (lazy loading)
* Environment: production
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Debug mode: on
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
* Restarting with stat
20/08/25 15:59:02 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
20/08/25 15:59:03 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
* Debugger is active!
* Debugger PIN: 131-873-430

Figure 9-4. *Flask API ready*

We also change the `Dockerfile` to copy the `app.py` file to the `/deploy` directory. In the last module, we run the `run.py` file. In this module, it is taken care of by the `app.py` file. Here is the updated `Dockerfile` content:

```
FROM jupyter/pyspark-notebook:latest
WORKDIR /deploy/
COPY ./requirements.txt /deploy/
RUN pip install -r requirements.txt
EXPOSE 5000
# Copy necessary files from local to container dir
COPY ./file.pkl /deploy/
COPY ./run.py /deploy/
COPY ./helper.py /deploy/
COPY ./assembleModel.h5 /deploy/assembleModel.h5
COPY ./char_label_model.h5 /deploy/char_label_model.h5
COPY ./clf_model.h5 /deploy/clf_model.h5
```

```
COPY ./app.py /deploy/
# Specify the entry point command to run when the container starts
ENTRYPOINT ["python", "app.py"]
```

Okay, we are set to perform real-time scoring. We will code the UI shortly. Let's first test the code using the Postman application.

Re-create the Docker image using the following code:

```
docker build -t scoring_image .
```

You don't need volume mapping this time, since you are going to perform live scoring. You need to make sure that the port number of the local system is mapped to the Docker port number.

```
docker run -p 5000:5000 scoring_image:latest
```

Once you get this message, you can switch back to the Postman API and test this app. You can download Postman online and install it if you don't have it already. Initiate the application and do the necessary changes, as shown next.

Postman API

This section is intended for readers who don't have a background in the Postman API (see Figure 9-5). You can skip this section if you are already familiar with it.

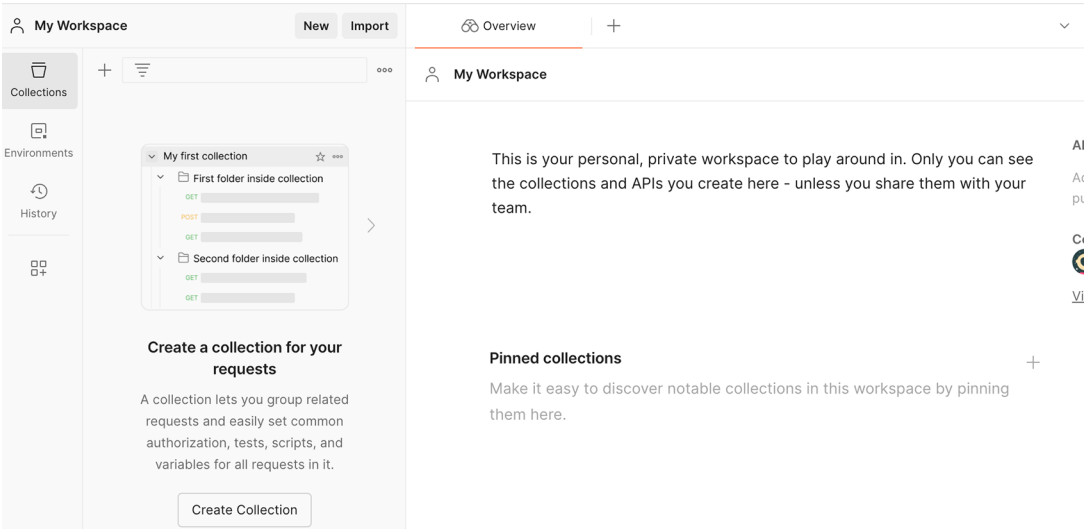

My Workspace
New
Import
Collections
Environments
History
My first collection
First folder inside collection
Second folder inside collection
Create a collection for your requests
A collection lets you group related requests and easily set common authorization, tests, scripts, and variables for all requests in it.
Create Collection
Overview
My Workspace
This is your personal, private workspace to play around in. Only you can see the collections and APIs you create here - unless you share them with your team.
Pinned collections
Make it easy to discover notable collections in this workspace by pinning them here.

Figure 9-5. *Postman API*

You need to create an API instance by clicking the New button in the left corner and then clicking Request Create a Basic Request to get the window shown in Figure 9-6.

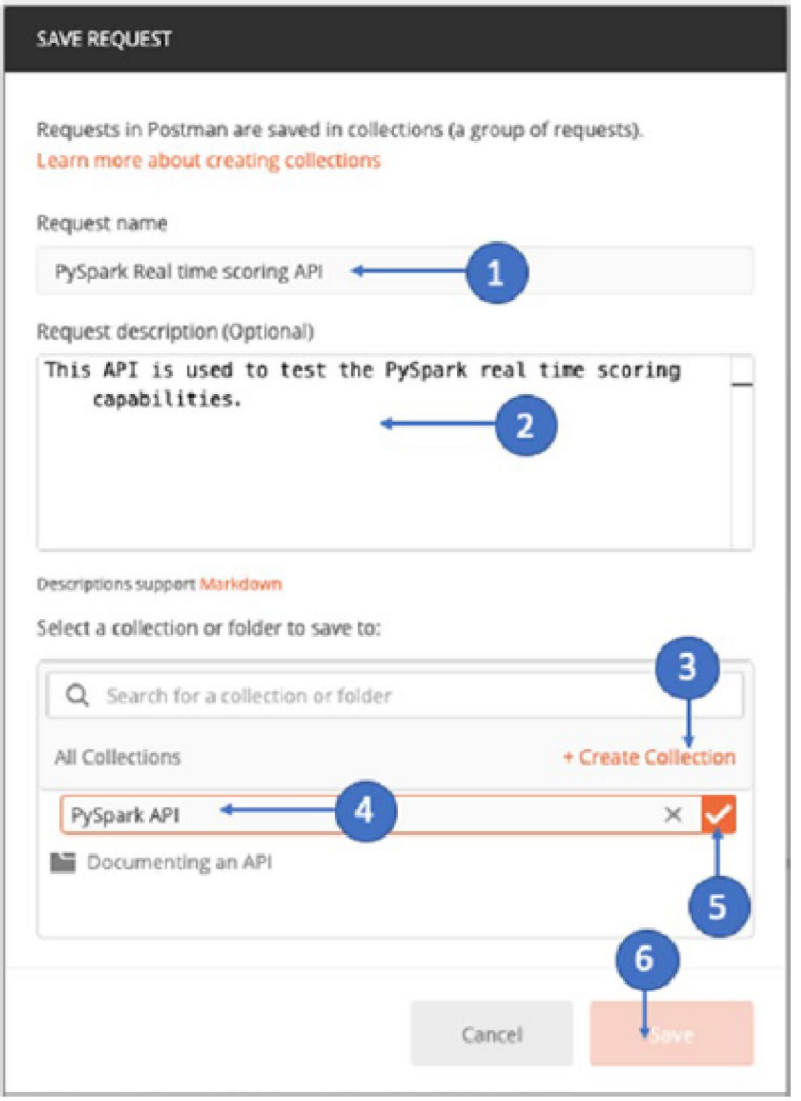


Figure 9-6. *Create a new API request*

Now, let's make the configurations to the API shown in Figure 9-7.

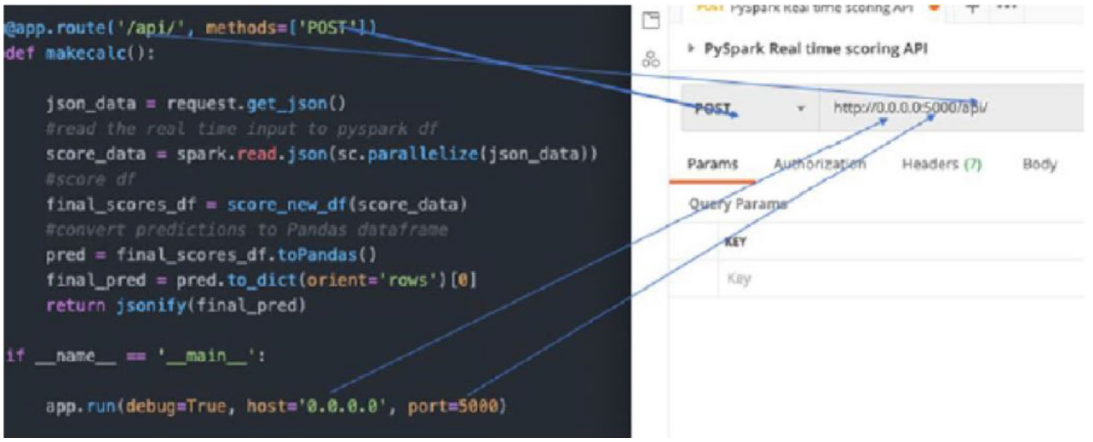


Figure 9-7. *Configurations*

Let's add a header column by navigating to the Headers tab. With this option, the API knows that it is looking for a JSON input (see Figure 9-8).

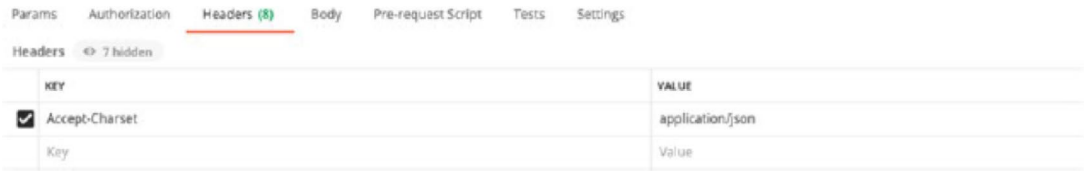


Figure 9-8. *JSON header object*

Finally, you are ready to score. Since you are providing the input for testing, switch the body type to `raw` and select JSON as the text option (see Figure 9-9).

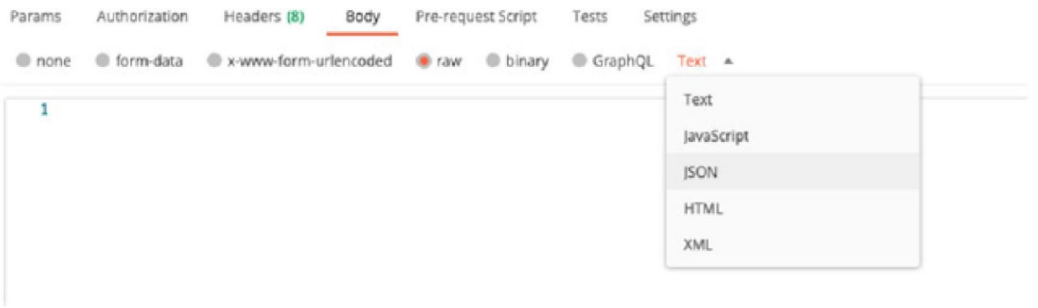


Figure 9-9. *JSON selection*

Test Real-Time Using Postman API

Use the following JSON object for testing purposes. Note the square brackets in front. This is used to pass a list of JSON objects.

```
[{"age":58,"job":"management","marital":"married","education":"tertiary",
"default":"no","balance":2143,"housing":"yes","loan":"no","contact":"unknown",
"day":5,"month":"may","duration":261,"campaign":1,"pdays":-1,"previous":0,"
poutcome":"unknown"}]
```

Copy and paste the JSON object into the body of the API and click Send (see Figure 9-10).

Figure 9-10. *Real-time scores*

You can now proceed to build the UI component for the app.

Build UI

We will use a Python package `streamlit` to build the user interface (UI). `streamlit` is easy to integrate with the existing code. One of the greatest advantages of using `streamlit` is that you can edit the Python code and see live changes in the web app instantly. Another advantage is that it requires minimal code to build the UI. Before we get started, take a look at the directory structure shown in Figure 9-11. The parent directory is `real_time_scoring`. We created two child directories: `pysparkapi` and `streamlitapi`. We will move all the code discussed so far to the `pysparkapi` directory. The UI code will reside in the `streamlitapi` directory.

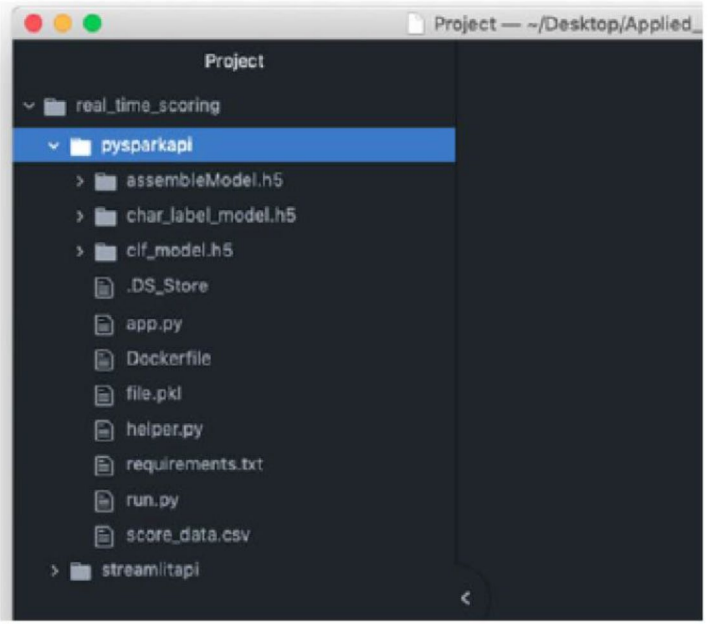


Figure 9-11. *UI directory structure*

The streamlitapi Directory

Navigate to the `streamlitapi` directory and create the following three files within it.

`Dockerfile` – *Docker for the frontend*

`requirements.txt` – *The required packages*

`webapp.py` – *The UI code*

You can test this code before deploying in a Docker container by running the following code:

```
streamlit run webapp.py
```

As mentioned, the interactive feature of `streamlit` lets you design custom layouts on the fly with minimal code. Okay, you are all set. These three files reside in the `streamlitapi` directory. You now have two Docker containers—one for PySpark Flask and the second one for the `streamlit` UI. You need to create a network between these two containers so that the UI interacts with the backend PySpark API. Let's switch to the parent directory, called `real_time_scoring`.

real_time_scoring Directory

A `docker-compose` file is used to combine multiple Docker containers and create a Docker network service. Let's look at the contents of this file.

docker-compose.yml

```
# Defining two services- pysparkapi, streamlitapi
version: '3'
services:
  pysparkapi:
    build: pysparkapi/
    ports:
      - 5000:5000
    networks:
      - deploy_network
    container_name: pysparkapi
  streamlitapi:
    build: streamlitapi/
    depends_on:
      - pysparkapi
    ports:
        - 8501:8501
    networks:
      - deploy_network
    container_name: streamlitapi
networks:
  deploy_network:
    driver: bridge
```

Executing the docker-compose.yml File

Now use the following code to execute the `docker-compose.yml` file:

```
docker-compose build
```

The Docker network service is established. You can now execute the Docker service using the code shown in Figure 9-12. It will take time to spin up the service; wait for the debugger pin to appear before you switch to your browser.

Sunders-MacBook-Pro:real_time_scoring sunder$ docker-compose up
Creating network "real_time_scoring_deploy_network" with driver "bridge"
Creating pysparkapi ... done
Creating streamlitapi ... done
Attaching to pysparkapi, streamlitapi
streamlitapi |
streamlitapi | You can now view your Streamlit app in your browser.
streamlitapi |
streamlitapi | Network URL: http://172.19.0.3:8501
streamlitapi | External URL: http://73.81.15.250:8501
streamlitapi |
pysparkapi | 20/08/25 23:19:53 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
pysparkapi | Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
pysparkapi | Setting default log level to "WARN".
pysparkapi | To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
pysparkapi | * Serving Flask app "app" (lazy loading)
pysparkapi | * Environment: production
pysparkapi | WARNING: This is a development server. Do not use it in a production deployment.
pysparkapi | Use a production WSGI server instead.
pysparkapi | * Debug mode: on
pysparkapi | * Running on http://0.0.0.0:6000/ (Press CTRL+C to quit)
pysparkapi | * Restarting with stat
pysparkapi | 20/08/25 23:20:12 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
pysparkapi | Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
pysparkapi | Setting default log level to "WARN".
pysparkapi | To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
pysparkapi | 20/08/25 23:20:13 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
pysparkapi | * Debugger is active!
pysparkapi | * Debugger PIN: 185-089-171

Figure 9-12. *Docker service creation*

Real-time Scoring API

To access the UI, you need to insert the following link in a browser:

```
http://localhost:8501/
```

The UI should look like Figure 9-13, if you used the same code as instructed in this book. We have shown only the partial UI in the figure. You can input the values for the features and click Get Predictions to get results.

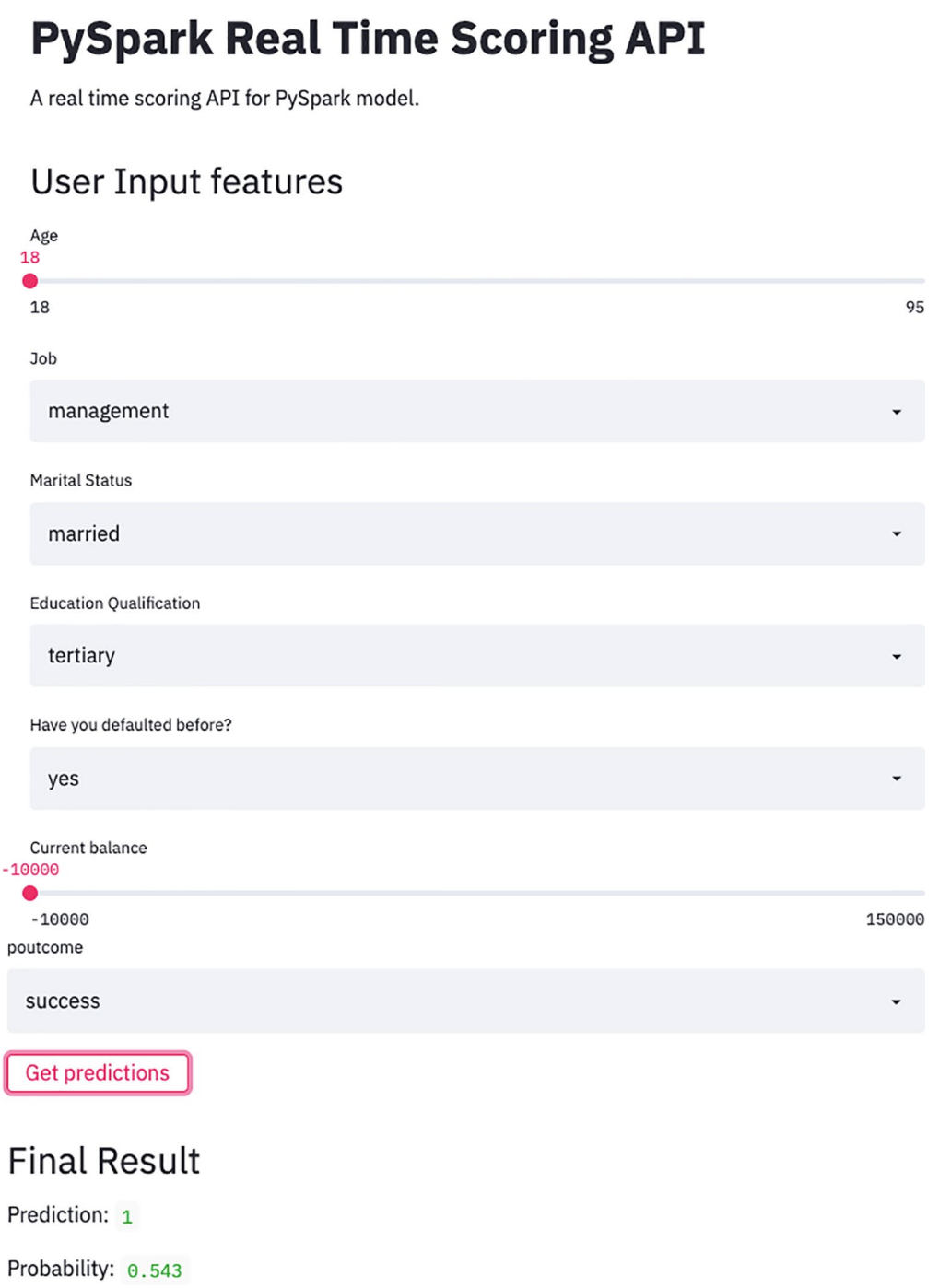


Figure 9-13. *Streamlit API*

When you want to kill this service, you can use the following code (see Figure 9-14):

```
docker-compose down
```

```
Sundars-MacBook-Pro:real_time_scoring sundar$ docker-compose down
Stopping streamlitapi ... done
Stopping pysparkapi   ... done
Removing streamlitapi ... done
Removing pysparkapi   ... done
Removing network real_time_scoring_deploy_network
```

Figure 9-14. *Killing the Docker service*

That's it. You have deployed the model in real-time with a nice frontend. The best part of this deployment is that the UI and PySpark sit in separate Docker containers. They can be managed and customized separately and can be combined with a `docker-compose` file when needed. This makes the software development lifecycle process much easier.

Summary

- You read about the model deployment process in detail.
- You deployed a model using both batch and real-time processing.
- Finally, you created a custom UI frontend using the `streamlit` API and integrated it with the PySpark API service using `docker-compose`.

Great job! In the next chapter, you learn about experimentations and PySpark code optimizations. Keep learning and stay tuned.

CHAPTER 10

Experimentation and Optimization Strategies

In this chapter, we dive deep into experimentation concepts. We also explore how to formulate clear research questions and hypotheses, and then use PySpark and its distributed computing framework to conduct scalable experiments on big data.

Importance of Experimentation

- **Unveiling Cause and Effect**: Using experimentation, you can establish causal relationships between variables and move beyond just observational data. In simple terms, it helps you understand not just "what" happens, but also "why" it happens.
- **Data-Driven Decision Making**: It allows you to gather data-driven insights, enabling you to optimize different strategies and make informed decisions.
- **Continuous Improvement**: It helps you to ensure the models and applications remain effective over time by iteratively testing and refining different approaches.

The following sections explore how PySpark specifically enhances experimentation capabilities in applied data science.

© Ramcharan Kakarla, Sundar Krishnan, Balaji Dhamodharan and Venkata Gunnu 2024
R. Kakarla et al., *Applied Data Science Using PySpark*, https://doi.org/10.1007/979-8-8688-0820-3_10

The Experimentation Process

Let's start with the high-level steps needed to conduct a successful experiment.

- **Define the Problem**: This is the question you are trying to address or the hypothesis you want to test. This is the foundational step in the experimentation process.
- **Design the Experiment**: Select the type of experiment that can help answer your question. There are multiple types of experiments: controlled experiments, field experiments, natural experiments, quasi-experiments, A/B testing, factorial experiments, longitudinal studies, and cross-sectional studies.
- **Identify Your Variables and Collect Data**: You need to clearly define your independent variable, dependent variable, and the variables that control the experiment. You need to decide how you will manipulate and measure them. Finally, you need to collect the data to test your hypothesis and decide how you will collect it. This can involve setting up pipelines in the appropriate way and storing them for future analysis.
- **Conduct the Experiment**: Run your experiment as per the design. You must control for *confounding variables* (variables that influence both your treatment and outcomes) and ensure that the experiment is conducted in a fair and ethical manner.
- **Analyze the Results**: Use statistical methods to analyze the results of your experiment. This can involve calculating means, running regression analyses, testing for statistical significance, and so on.
- **Interpret and Communicate Results**: Draw conclusions from your analysis and communicate your findings. This can involve writing a report, presenting your results to stakeholders, and publishing your findings in a scientific journal.
- **Refine and Iterate**: Based on the results of your experiment, you may need to refine your question, design a new experiment, or collect more data. Experimentation is often an iterative process.

Hypothesis Testing

A *hypothesis* is a claim or assertion that can be tested. Using hypothesis testing, you can either reject or leave unchallenged a particular statement. This may be more intuitive with an example. An example of a claim is people who eat breakfast are less prone to diabetes than people who don't eat breakfast. Another example is, does churn affect the population who receive an offer versus those who don't? The idea is to check if these assertions are true for the population. There are two types of hypotheses.

- **Null hypothesis:** This is represented by *H0*. It is based on conventional wisdom or experience.
- Alternate Hypothesis: This is represented by *Ha*. This is often the hypothesis of interest that you are trying to validate.

Let's say you have to formulate the hypothesis for the breakfast example. You can write the hypothesis as follows:

H0 : Eating breakfast has no effect on the diabetic rate

Ha: Eating breakfast has a negative effect on the diabetic rate

The p-value is used as an alternative to rejection points to provide the smallest level of significance at which the null hypothesis would be rejected. A smaller p-value signifies stronger evidence in support of the alternative hypothesis.

The following are the types of errors:

Actual	H0 is True	H0 is False
Decision		
Fail to reject null	Correct	Type II error
Reject null	Type I error	Correct

The probability of a Type I error is denoted by α, and the probability of a Type II error is denoted by β. The power of statistical test is given by $1-\beta$. Depending on the use case, focus on reducing α or β. Most textbooks focus on reducing α, since the rationale is to stick to the conventional approach (at least not making things worse with the alternate approach).

Consider the example of flipping a coin. Effect size can be defined as the difference between heads and tails for *n* flips. Sample size on the other hand is the number of heads and tails. Effect size and sample size both influence the p-value. This is why you

can't reduce both Type I and Type II errors simultaneously. In hypothesis testing, you first formulate the null hypothesis and then the alternate hypothesis. Next, you decide the desired level of significance α (usually 5%) for most tests. Then you run the test and decide based on the significance level. If the p-value is less than the significance level, you reject the null hypothesis. If the p-value is more than or equal to the significance level, you fail to reject the null hypothesis. The PySpark statistics package provides built-in methods to run Pearson's chi-squared tests and the Kolmogorov-Smirnov test. It does not support most of the hypothesis testing methods. However, they can be coded using user defined functions. These are listed in the following table.

When to Use?	Parametric	Non-Parametric
Compare means of two groups	T-test	Mann-Whitney U Test
Compare means of two related groups	Paired t-test	Wilcoxon signed ranked test
Compare means of more than two groups	One-way Anova	Kruskal-Wallis Test
Assess relationship between two variables	Pearson Correlation	Spearman Correlation

Experimentation Case Study

Let's pick some example data to learn how to perform A/B testing with PySpark. Imagine you're running an e-commerce company and you think you could improve the website to engage customers more and thereby improve conversion rates. You have data about the customers and their actions on the website. The sample data is shown here:

user_id	timestamp	group	landing_page	converted
851104	2017-01-21 22:11:48.556739	control	old_page	0
804228	2017-01-12 08:01:45.159739	control	old_page	0
661590	2017-01-11 16:55:06.154213	treatment	new_page	0
853541	2017-01-08 18:28:03.143765	treatment	new_page	0
864975	2017-01-21 01:52:26.210827	control	old_page	1
936923	2017-01-10 15:20:49.083499	control	old_page	0
679687	2017-01-19 03:26:46.940749	treatment	new_page	1
719014	2017-01-17 01:48:29.539573	control	old_page	0

You can get this sample data from Kaggle at https://www.kaggle.com/datasets/zhangluyuan/ab-testing/data.

The dataset has five columns:

User_Id – Unique ID for each user.

Timestamp – When the user visited the site.

Group – Whether the user is assigned to the treatment or the control group. Users assigned to the treatment group will see the new site, whereas the control group users will see the old site.

Landing_page – Description of whether the user saw the `new_page` or the `old_page`.

Converted – Whether the user made a transaction represented as 0 (no) and 1 (yes).

Let's review the experimentation process discussed at the beginning of the chapter and see how it can be applied to this A/B testing case study.

- **Define the Problem:** Improve the website's conversion rate. Conversion rate is defined as the "total number of purchases" divided by the "total number of actions". A user can take three actions on the website: `view_product`, `add_to_cart`, and `purchase`.
- **Design the Experiment:** You want to improve the website and compare its performance to the existing website. This is a classic A/B testing use case, so we will stick to it.
- **Identify Your Variables and Collect Data:** In this experiment, you will randomly allocate people to control and test (50/50). You can potentially control for confounders (age, gender), but those are excluded from the chapter. Therefore, the independent variable for this experiment is whether the user received treatment (test, saw the new website) or not(control, saw the old website). The dependent variable for this exercise is the purchase rate. There are no control variables.
- **Conduct the Experiment:** Assume that the experiment is run fairly and that the population assigned for treatment is selected randomly.

- **Analyze the Results:** You can use a t-test to analyze the experiment's results. But wait, you don't have a Spark package to do the t-test out of the box. We will show you two ways to run this experiment and analyze the means of the outcome.
- **Interpret and Communicate Results:** Finally, you make interpretations based on the results and then make decisions.
- **Refine and Iterate:** You can change the website layout again and start the whole process all over.

Before diving into the A/B test, consider some high-level stats about the data.

```
from pyspark.sql.functions import col, count

data.groupby(['group', 'landing_page']).count().show()
```

```
+---------+------------+------+
|    group|landing_page| count|
+---------+------------+------+
|treatment|    new_page|145311|
|  control|    new_page|  1928|
|treatment|    old_page|  1965|
|  control|    old_page|145274|
+---------+------------+------+
```

Something is not right with this data. It seems like some users landed in the wrong `landing_page`. This might be due to some noise during the start of the experiment or some other external factors. Let's clean up the data to filter for the correct treatment and `landing_page` assignment.

```
# filter appropriate data
data = data.where(((data.group ==  'treatment') & (data.landing_page=='new_
page')) |
           ((data.group ==  'control') & (data.landing_page=='old_page')))
Let's check our data once again.
data.groupby(['group', 'landing_page']).count().show()
```

```
+---------+------------+------+
|    group|landing_page| count|
+---------+------------+------+
|treatment|    new_page|145311|
|  control|    old_page|145274|
+---------+------------+------+
```

Now, let's check the average outcomes. This helps you determine how many in the `control` group make purchases and how that differs from the `treatment` group:

```
# check group means
data.groupby(['group', 'landing_page']).agg(avg("converted")).show()
```

```
+---------+------------+--------------------+
|    group|landing_page|     avg(converted)|
+---------+------------+--------------------+
|treatment|    new_page|0.11880724790277405|
|  control|    old_page| 0.1203863045004612|
+---------+------------+--------------------+
```

Another way to calculate the same thing is to figure out the `uplift` percentage:

```
# Split the DataFrame into two: one for the control group, one for the
treatment group
control_df = data.filter(data.group == "control")
treatment_df = data.filter(data.group == "treatment")

# Calculate conversion rates for each group
control_conversions = control_df.filter(col("converted") == 1).count()
control_rate = control_conversions / control_df.count()

treatment_conversions = treatment_df.filter(col("converted") == 1).count()
treatment_rate = treatment_conversions / treatment_df.count()

# Calculate uplift (optional)
uplift = (treatment_rate - control_rate) / control_rate * 100

# Print results
```

```
print("Control Group Conversion Rate:", round(control_rate, 4))
print("Treatment Group Conversion Rate:", round(treatment_rate, 4))
print("Uplift (if applicable):", round(uplift, 4), "%")
```

```
Control Group Conversion Rate: 0.1204
Treatment Group Conversion Rate: 0.1188
Uplift (if applicable): -1.3117 %
```

Remember the conversion rate averages. You will use this later in the chapter. Note the -1.3% uplift in the conversion because of the new website layout. Can you conclude that the new layout is not working out as expected and continue with the old layout for all members going forward? In this case, you can, since the conversions are not trending in the right direction. However, you might see some use cases where the conversions are slightly higher than the control group, so you are still inconclusive about making the change. To answer this question conclusively, you need to find evidence. In other words, you need to reject the null hypothesis using t-test. Assuming, in this case, you are still interested in pursuing the new layout, let's conduct a t-test to see what we learn from there.

H0: Mean of purchase (old website layout) = Mean of purchase (new website layout)

H1: Both means are different

We need evidence to reject H0 and accept H1. So, let's run a t-test using Python.

```
# this code creates a list of the outcome variable
# for each treatment type

control_data_flag = control_df.select('converted').rdd.flatMap(lambda x:
x).collect()
treatment_data_flag = treatment_df.select('converted').rdd.flatMap(lambda
x: x).collect()

import scipy.stats as stats

# Calculate the t-statistic and p-value
t_statistic, p_value = stats.ttest_ind(treatment_data_flag, control_
data_flag)

# Print the results
print("t-statistic:", t_statistic)
```

```
print("p-value:", p_value)
t-statistic: -1.31
p-value: 0.19
```

Notice that the t-test results are not significant. This means that you fail to reject the null hypothesis. Therefore, you can safely conclude that the new iteration of the website layout does not significantly drive more purchases compared to the old website layout. Although this experiment did not yield a statistically significant result, it gives hope that further iterations can be tested in the future, which could potentially improve purchase behavior.

Separately, we also provided a fake sample use case in a separate notebook in GitHub. When you run that experiment, what you will notice is that it shows an uplift of 6%. In this scenario, the t-test also shows insignificant results. Unfortunately, when you fail to reject null hypothesis, you don't have significant evidence that the new version is better than old. Therefore, you need to investigate further, refine your analysis, and iterate the experiment.

Now, let's see how you can run the t-test in PySpark using Spark scalable capabilities. Remember, when you learned about the linear regression in Chapter 5, you looked at the variable coefficients and the p-value significance of the variable. These significant p-values are based on the t-test. So, let's leverage that and see if you get the same result using linear regression. To run a linear regression, you need to convert the input features to vectors.

```
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorAssembler

vectorAssembler = VectorAssembler(inputCols=["group_flag"], outputCol="v_
group_flag")
v_data = vectorAssembler.transform(data)

# Define the linear regression model
lr = LinearRegression(featuresCol='v_group_flag', labelCol='converted')

# Fit the model
lr_model = lr.fit(v_data)

# Print the coefficients and intercept for linear regression
```

```
print("Coefficients: " + str(lr_model.coefficients))
print("Intercept: " + str(lr_model.intercept))
```

```
Coefficients: [-0.0016]
Intercept: 0.1204
```

The regression intercept is the same as the `Control` group conversion rate, as you calculated before. The coefficient, -0.0016, is the incremental conversion driven by the treated group, which equals (0.1204 - 0.0016 = 0.1188). The values match the previous calculated values.

```
print(lr_model.summary.tValues)
print(lr_model.summary.pValues)
[-1.31, 141.41]
[0.19, 0.0]
```

We get the t-statistic and p-values using the previous code. The first value corresponds to the treatment and the second one corresponds to the intercept; in this case (control). So, the t-statistic value from the regression is same as the t-statistic calculation that you did before. Also, the p-values match the previous calculation. This method demonstrates a scalable approach to conducting a t-test in PySpark.

This is a simple example to showcase the experimentation capabilities using PySpark. Of course, when you use real-world data, you need to consider adding necessary features (confounders, instrumental variables, and omitted variable bias) and data collection methods (control variables and sampling strategies), and finally, making sure that the experiment is clean and reproducible.

You can expand this experiment to measure the effect of treatment on outcome for multiple time periods. You need to measure the input features for the same time period along with the observed outcomes, which lets you construct panel data. With that, you can perform a difference-in-difference regression to analyze the treatment effect over a period. This concept is out-of-scope for this book.

Why Use PySpark for Experimentation?

- The primary advantage of choosing PySpark is its scalability and efficiency. Spark can handle terabytes or even petabytes of data, whereas other tools need additional infrastructure to support the same level of scalability. PySpark natively parallelizes its operations and

therefore is highly efficient to manipulate data and train models. This becomes helpful when working with real-time streaming data use cases.

- PySpark provides MLlib and SQL packages for model training and data wrangling. It assists in seamless integration with both libraries, simplifying data cleaning tasks for experimentation.
- PySpark can work with traditional data formats (CSV, JSON, and parquet) and streaming sources (Kafka). Thus, it is easy to switch to diverse sources without worrying about format limitations. Second, it utilizes Python and can, therefore, easily integrate with existing Python-based data workflows.
- PySpark ensures the deterministic execution of your experiments; therefore, the same code will produce the same results on the same data regardless of cluster configuration. This helps you reproduce results and facilitates collaboration with other data scientists. In addition, you can also share your notebooks with other data scientists for code reproducibility.
- For organizations with limited budgets, this is one of the cost-effective open-source solutions available on the market for performing experimentation on large-scale data.

Optimization Strategies with PySpark

Random Data Generation

You might often want to create data to test scenarios and algorithms. PySpark has a handy built-in function that can do that by calling simple functions:

```
from pyspark.mllib.random import RandomRDDs
# Generate a random double RDD that contains 1 million i.i.d. values drawn
# from the standard normal distribution `N(0, 1)`, evenly distributed
# in 10 partitions.
u = RandomRDDs.normalRDD(spark.sparkContext, 1000000, 10)
# Apply a transform to get a random double RDD following `N(1, 4)`.
v = u.map(lambda x: 1.0 + 2.0 * x)
```

This example generates a random double RDD, whose values follow the standard normal distribution N(0, 1), and then maps it to N(1, 4).

Sampling

Sampling is an important concept in machine learning. If the sample dataset is not representative, experiments with biased measures often carry little to no value. This section introduces the types of sampling available in Pyspark.

Simple Random Sampling (SRS)

In simple random sampling, every row is randomly selected and has an equal chance of being picked. `sample()` aids in performing SRS. You have a couple of options within SRS. For demonstration, we use the churn modeling dataset introduced in Chapter 8.

SRS with replacement:

```
# Read data
file_location = "churn_modelling.csv"
file_type = "csv"
infer_schema = "false"
first_row_is_header = "true"
delimiter = ","

df = spark.read.format(file_type)\
.option("inferSchema", infer_schema)\
.option("header", first_row_is_header)\
.option("sep", delimiter)\
.load(file_location)

# Simple Random Sampling without replacement
df=df.select('Geography','NumOfProducts','Age','Gender','Tenure','Exited')
df_srs_without_rep = df.sample(False, 0.5, 23)
df_srs_without_rep.show(10,False)
```

SRS without replacement:

```
# Simple Random Sampling with replacement
df_srs_without_rep = df.sample(True, 0.5, 23)
df_srs_without_rep.show(10,False)
```

Stratified Sampling

With stratified sampling, every member of the population is grouped into homogeneous subgroups called *strata* and a representative of each group (strata) is chosen.

```
# Stratified Sampling


from pyspark.sql.types import IntegerType
df = df.withColumn("Exited", df["Exited"].cast(IntegerType()))
df.select('Exited').describe().show()
# Perform stratified sampling based on column "Exited"
stratified_sampled = df.sampleBy("Exited", fractions={0: 0.5, 1: 0.5}, seed=23)
stratified_sampled.select('Exited').describe().show()
```

The following provides the raw data (left) and stratified sample distributions (right):

```
+-------+------------------+
|summary|            Exited|
+-------+------------------+
|  count|             10000|
|   mean|            0.2037|
| stddev|0.40276858399486065|
|    min|                 0|
|    max|                 1|
+-------+------------------+

+-------+------------------+
|summary|            Exited|
+-------+------------------+
|  count|              5022|
|   mean| 0.2078853046594982|
| stddev|0.40583469642655867|
|    min|                 0|
|    max|                 1|
+-------+------------------+
```

Difference Between the coalesce and repartition Functions

The `coalesce` and `repartition` functions are both used to control the number of partitions while saving the data. Here is a quick tip: When archiving small and medium size data, use `coalesce` over `repartition` as a quick solution. Otherwise, it is best to use `repartition`. Here is why:

- `coalesce` uses existing partitions to minimize the data shuffling, resulting in uneven partition sizes. It's efficient for consolidating data into fewer partitions without a full shuffle.
- `Repartition` does a full shuffle and creates a new partition. This ensures equal-size partitions of data. It is also recommended that you repartition your data after running filtering queries to improve the efficiency of your queries. If you are not repartitioning your data after filtering, you may end up with many memory partitions, resulting in a poor performance. You also have to keep in mind the cost of a full shuffle when repartitioning data.

Directed Acyclic Graph DAG scheduler gives nice insight into how each operation is applied to data.

Switching between Python and PySpark

PySpark is best suited for large datasets, whereas pandas is faster for smaller datasets. You can use pandas DataFrames with PySpark DataFrames in multiple situations to optimize the execution times of your code. You saw a classic example in Chapter 8, where you built automated pipelines. For calculating metrics, we used PySpark to do the heavy lifting. And once we had manageable data (rows in 10 to 100's with few columns), we moved them to pandas to efficiently process and calculate metrics.

This is a great strategy when you know you will be working with rolled data cubes. You can process the large datasets in PySpark and convert the grouped PySpark DataFrames into pandas DataFrames. Since pandas is more mature than PySpark, it will open up a myriad of packages that can be useful in various situations.

Curious Character of Nulls

In PySpark there are situations when you are trying to fill a null value in a column and you use the `fillna` function. `fillna()` works fine, but you still end up with nulls in columns. Yes, you guessed it right, when the datatype of the column you are trying to fill doesn't match the `fillna` value, you end up with a null value. For example, calculating the mean of an integer column can result in a `float` value. Now, if you are trying to fill using this `float` value, you may still observe nulls. The workaround is to ensure that the column's datatype matches the datatype of the value you intend to use for filling nulls.

Common Function Conflicts

There will be instances when functions such as `max` exist in both Python and PySpark SQL functions. There will be an import conflict error if you use a wildcard import from `pyspark.sql.functions`.

Join Conditions

Joins are one of the key operations of Spark programs. In general, Spark has two cluster communication strategies. The node-to-node strategy involves Spark data shuffles across the clusters. Per the node, the strategy performs broadcast joins. There are three main types of joins that Spark uses:

- **Sort Merge Join:** It contains a couple of steps. The first step is to sort the datasets and the second operation is to merge the sorted data in the partition by iterating over the elements by the join key. This is usually the default setting in Spark. To achieve an optimized sort merge, the joining DataFrame should be uniformly distributed on the joining columns. If all the join keys are co-located, the join will be further optimized, as there is no need for shuffle operations. The sort merge join is preferred when both sides are large.
- **Broadcast Join:** These joins are the most efficient ones in Spark. They are useful for joining a small dataset to a huge dataset. In this join, a smaller table will be broadcast to all worker nodes. Always make sure that you are broadcasting the smaller table.

- **Shuffle Hash Join:** This join is based on the concept of MapReduce. This process maps through the DataFrames and uses the values of the join column as the output key. Spark uses a custom function to ensure that the Shuffle Hash join is better for the given dataset than the broadcast join. Creating a hash table is an expensive operation.

It is always a good practice to verify if any nulls exist in the join key. This mistake can be expensive. It might lead to unrelated errors, such as errors occurred while calling `o206.ShowString` or `o64.cacheTable`. Another reason can be duplicates. If nulls and duplicates are checked before joining tables, you can save some time by not running into any of the previously mentioned errors.

There are two major strategies Spark uses for joins. Use a sort merge join when both sides are large and a broadcast join when one side is small.

User Defined Functions (UDFs)

You may encounter issues with UDF even after successfully importing it into the running file. Again, you might encounter a strange `ShowString` error. The real reason is that UDF is not available to some workers. You can overcome this error by adding the following line to your script. It enables other workers to reference your module file.

```
sc.addPyFile(path_to_your_module.py)
```

It is also recommended that your UDF usage be minimal. This is due to the overhead of the deserialization to reserialization operation that it triggers in the backend.

Handle the Skewness

Imbalanced datasets can cause spikes in execution duration. You can observe this in the Spark UI backend from the `min` and `max` time values. Using evenly distributed datasets can significantly improve the performance.

In a SQL join operation, the join key is changed to redistribute data evenly so that processing for a partition does not take more time. This technique is called *salting*.

Add a column to each side with a random `int` between 0 and `` `spark.sql.shuffle.partitions` ``-1 to both sides and then add a `join` clause to include a join on the generated column.

Using Cache

Cache is a great tool in the arsenal to speed up performance. If you use cache for all the objects, it can end up in slightly slower storage. Using cache takes away some of your session's processing capability. In the Storage tab, you can observe the size of memory and the size of disk distribution.

Persist/Unpersist

You have learned that Spark uses lazy evaluation to execute commands. Persist is one of the attractive options available in the Spark store. It provides the ability to store intermediate RDD around the cluster for faster access. For illustration, say you have an initial dataset that you read into Spark and need to apply a filtering operation. Since Spark works on the concept of lazy evaluation, filtered data doesn't contain the actual data. It just maintains the order of DAG that needs to be executed for transformation to create the filtered data. If the filtered dataset is going to be consumed by different objects to generate different results, for every single iteration, the transformation occurs on the filtered dataset. Imagine there are 100 operations on this filtered dataset. This can quickly add to the processing time and stretch the operation from a few minutes to hours for a sizable data.

Persisting data can save a lot of computational time. When the filtered data is calculated for the first iteration, it is persisted in the memory. In subsequent runs, this DataFrame in the memory is used for all the operations downstream. After your required downstream operations are completed, it is good practice to unpersist the stored data from memory and disk.

Shuffle Partitions

These partitions exist when shuffling data for join or aggregations. Whenever shuffling, huge chunks of data are moved between the partitions. This shuffle can happen within the executor or across the executors. These operations can get expensive very quickly. Imagine you have a small DataFrame with eight initial partitions. By doing a simple group by operation, you shoot the shuffle partition count up to 200. It is because of the

default shuffle that partition for the DataFrame is set to 200. You have the opportunity to reset the partitions using these configurations:

```
# Set the number of shuffle partitions
sparkSession.conf.set("spark.sql.shuffle.partitions",50)
```

A few additional settings to explore for shuffling operations:

```
# Set the maximum size of reduce data in flight during shuffling
sparkSession.conf.set("spark.reducer.maxSizeInFlight", "96MB")
# Enable or disable compression sparkSession.conf.set("spark.shuffle.
compress", "true")
# Set the buffer size to writing shuffle files to disk sparkSession.conf.
set("spark.shuffle.file.buffer", "1m")
```

If you are not setting these configurations, you may underutilize your Spark resources.

Use Optimal Formats

Spark supports many formats, including CSV, JSON, XML, parquet, orc, and Avro. The best format for performance is parquet with snappy compression, which is the default from Spark 2.x or higher. Parquet stores data in columnar format and is highly optimized for efficient data processing in Spark.

Data Serialization

Spark jobs are distributed, so appropriate data serialization is important for the best performance. There are two serialization options for Spark:

- Java serialization is the default option.
- Kryo serialization is a newer format and can result in faster and more compact serialization than Java. Kryo requires that you register the classes in your program, and it doesn't support all Serializable types.

Accomplishments

- You learned about the overall process of experimentation.
- You learned the overview of hypothesis testing and different sampling techniques.
- You learned some of the common tips and optimizations that can save time in day-to-day activities.
- You also had an overview of some performance fine-tuning techniques.

Fantastic job! Happy PySparking!!

CHAPTER 11

Modeling Frameworks

Predictive modeling has become an indispensable tool for businesses striving to gain a competitive edge in today's data-driven landscape. By leveraging historical data to forecast future outcomes, companies can make informed decisions that enhance customer satisfaction, optimize marketing strategies, and ultimately drive growth. However, the true power of predictive models lies in their ability to be integrated and stacked, creating comprehensive frameworks that deliver actionable insights and significant value to stakeholders.

The goal of this chapter is to delve into the integration of various predictive models to build robust frameworks that address critical business challenges. By understanding and applying these frameworks, organizations can improve their decision-making processes, leading to increased efficiency, profitability, and customer retention.

Customer Lifetime Value

Customer Lifetime Value (CLV) is a fundamental concept in business and marketing that quantifies the total revenue a company can expect to generate from a single customer account over the entire duration of their relationship. This metric has become increasingly important in today's competitive business landscape, where customer acquisition and retention are critical to long-term success. By understanding and leveraging CLV, businesses can make informed decisions about resource allocation, marketing strategies, and customer retention efforts, ultimately driving profitability and sustainable growth.

© Ramcharan Kakarla, Sundar Krishnan, Balaji Dhamodharan and Venkata Gunnu 2024
R. Kakarla et al., *Applied Data Science Using PySpark*, https://doi.org/10.1007/979-8-8688-0820-3_11

Understanding Customer Lifetime Value (CLV)

At its core, CLV represents the net profit a company can expect to earn from a customer throughout their entire relationship with the business. It encompasses all revenue streams from that customer, including initial purchases, repeat purchases, and any additional value they may bring through referrals or word-of-mouth marketing. CLV is a forward-looking metric that helps businesses anticipate future revenue streams and make strategic decisions based on the long-term value of their customers.

The basic formula for calculating CLV is:

CLV = (Average Purchase Value) * (Purchase Frequency)* (Customer Lifespan)

Where:

Average Purchase Value is the average amount spent by the customer per transaction. This value can be calculated by dividing the total revenue generated from a customer by the number of transactions they made.

Purchase Frequency is the average number of purchases made by the customer in each time period, such as annually or monthly. This metric can be derived by analyzing historical purchase data and identifying patterns in customer behavior.

Customer Lifespan is the average duration a customer continues to purchase from the business. This can be estimated based on historical data, industry benchmarks, or predictive models that consider factors such as customer churn rates and retention strategies.

This basic formula provides a simplified approach to calculating CLV, and more sophisticated models may incorporate additional factors to enhance accuracy. These factors can include customer acquisition costs, gross margins, discount rates to account for the time value of money, and customer-specific variables such as demographics, purchase history, and engagement levels.

Importance of CLV

Customer Lifetime Value is a critical metric for businesses for several reasons:

1. Resource Allocation

   CLV helps businesses allocate resources more effectively by identifying high-value customers. By focusing on retaining these customers, companies can maximize their return on investment (ROI) and ensure steady revenue streams. This is particularly

important in industries with high customer acquisition costs, where retaining existing customers is more cost-effective than acquiring new ones.

2. Customer Retention

   Understanding CLV allows businesses to develop targeted retention strategies. Since acquiring new customers is often more expensive than retaining existing ones, improving CLV can significantly reduce marketing and sales costs. By identifying and addressing the needs of high-value customers, businesses can foster loyalty and reduce customer churn, ultimately increasing their overall profitability.

3. Marketing Strategy

   CLV provides insights into the effectiveness of marketing campaigns. By comparing the CLV of different customer segments, businesses can tailor their marketing efforts to attract and retain the most profitable customers. This data-driven approach ensures that marketing resources are allocated efficiently and campaigns are optimized for maximum impact.

4. Financial Forecasting

   CLV is a valuable tool for financial forecasting. By estimating the future revenue streams from existing customers, businesses can make more accurate projections and informed decisions about budgeting, investment, and resource allocation. This long-term perspective is essential for strategic planning and ensuring the financial sustainability of the business.

5. Customer Segmentation

   By analyzing CLV, businesses can segment their customer base into different value tiers. This segmentation allows for personalized marketing and customer service strategies, enhancing customer satisfaction and loyalty. High-value customers can be targeted with premium offerings and personalized experiences, while lower-value segments can be approached with more cost-effective strategies.

Types of CLV

While the concept of CLV is straightforward, there are several approaches to calculating and analyzing this metric, each with its own strengths and applications. The following are some of the most common types of CLV:

1. Historical CLV

   Historical CLV measures the total revenue generated by a customer based on past transactions. This type of CLV is useful for understanding the value of existing customers and building profiles of ideal customers. By analyzing historical data, businesses can identify patterns and characteristics of their most valuable customers, which can inform future acquisition and retention strategies.

   However, historical CLV has limitations in predicting future revenue, as it does not account for changes in customer behavior, market conditions, or the business's own strategies. It provides a retrospective view but may not accurately reflect the potential value of a customer going forward.

2. Predictive CLV

   Predictive CLV uses historical data and predictive analytics to estimate the future value of a customer. This approach considers factors such as customer acquisition costs, purchase frequency, and business overheads to provide a more accurate prediction of future revenue. Predictive CLV is particularly useful for long-term strategic planning, as it helps businesses anticipate future cash flows and make informed decisions about resource allocation and investment.

   Predictive models can incorporate a wide range of variables, including customer demographics, purchase history, engagement levels, and external factors such as market trends and economic conditions. By leveraging advanced analytics techniques like machine learning and artificial intelligence, businesses can continuously refine their predictive models and improve the accuracy of their CLV estimates.

3. Cohort Analysis

   Cohort analysis groups customers based on the time period they made their first purchase and tracks their behavior over time. This method helps identify patterns in customer behavior and compare the CLV of different customer segments. By analyzing cohorts, businesses can gain insights into customer acquisition, retention, and revenue trends, allowing them to optimize their strategies accordingly.

   For example, a business could analyze the CLV of customers acquired through different marketing channels or during specific promotional periods. This information can help them understand which acquisition strategies are most effective in attracting high-value customers, guiding adjustments and enhancements to their marketing strategies.

4. Retention-Based CLV

   This method focuses on the number of repeat purchases, average purchase value, and customer retention rate. It is particularly useful for subscription-based businesses where customer retention is a key metric. By analyzing retention rates and the revenue generated from repeat customers, businesses can identify opportunities to improve customer loyalty and maximize the lifetime value of their customer base.

   Retention-based CLV models often incorporate factors such as customer churn rates, renewal rates, and the impact of retention strategies on customer behavior. This approach helps businesses understand the long-term value of retaining customers and prioritize initiatives that foster customer loyalty and reduce churn.

5. RFM Analysis

   RFM (Recency, Frequency, Monetary) analysis considers three factors: how recently a customer made a purchase, how often they purchase, and how much they spend. This method helps businesses identify high-value customers and tailor their marketing strategies accordingly.

By segmenting customers based on their RFM scores, businesses can prioritize their efforts and allocate resources more effectively. For example, customers with high recency and frequency scores may be targeted with loyalty programs or personalized offers, while those with low scores may require reactivation campaigns or win-back strategies.

RFM analysis is particularly useful in industries with high customer churn rates, as it helps businesses identify at-risk customers and implement proactive measures to retain them.

CLV in Different Business Models

While the concept of CLV is applicable across various industries, its implementation and specific considerations may vary depending on the business model. Here's how CLV is applied in different business contexts:

1. E-commerce

   In e-commerce, CLV helps businesses understand the long-term value of their customers and optimize their marketing spend. By focusing on high-CLV customers, e-commerce businesses can increase repeat purchases and customer loyalty. One of the key challenges in e-commerce is the high customer acquisition costs associated with online advertising and marketing campaigns. By identifying and retaining high-value customers, e-commerce businesses can offset these costs and maximize their profitability.

   CLV analysis in e-commerce often involves tracking customer behavior across multiple channels, including website visits, email interactions, and social media engagement. This data can be used to segment customers based on their purchase history, browsing patterns, and engagement levels, allowing for targeted marketing and personalized experiences. Additionally, e-commerce businesses can leverage predictive CLV models to identify customers with high potential value and tailor their acquisition and retention strategies accordingly. This may involve offering personalized recommendations, loyalty programs, or targeted promotions to encourage repeat purchases and increase customer lifetime value.

2. Subscription-Based Models

   For subscription-based businesses, CLV is crucial for understanding the value of recurring revenue. It helps these businesses optimize their pricing strategies, reduce churn, and increase customer retention. In subscription models, CLV is often calculated based on the average revenue per user (ARPU), customer acquisition costs, and customer churn rates. By analyzing these factors, businesses can identify the optimal pricing and retention strategies to maximize the lifetime value of their customer base.

   For example, a subscription-based business may offer discounts or incentives to customers who commit to longer subscription periods, as this can increase their CLV by reducing churn and ensuring a steady stream of recurring revenue. Additionally, subscription businesses can leverage CLV analysis to identify high-value customers and prioritize retention efforts. By understanding the characteristics and behaviors of their most valuable customers, these businesses can develop targeted retention strategies, such as personalized communication, exclusive offers, or enhanced customer support.

3. B2B (Business-to-Business)

   In B2B industries, CLV can be significantly higher due to the larger transaction sizes and longer sales cycles. B2B companies use CLV to prioritize high-value accounts and allocate resources effectively to nurture these relationships. CLV in B2B contexts often involves analyzing factors such as contract values, renewal rates, and cross-selling opportunities. By understanding the potential lifetime value of a customer account, B2B businesses can make informed decisions about resource allocation, account management, and customer support.

For example, a B2B company may prioritize account management resources for their highest-CLV customers, ensuring that these valuable relationships receive dedicated support and attention. Additionally, they may develop targeted cross-selling strategies to increase the lifetime value of these accounts by offering complementary products or services. B2B companies can also leverage CLV analysis to identify opportunities for account expansion or upselling. By understanding the needs and potential value of their customers, they can tailor their offerings and sales strategies to maximize revenue and profitability.

4. SaaS (Software as a Service)

   For SaaS companies, CLV is a key metric for understanding the profitability of their subscription models. It helps these companies optimize their customer acquisition costs and develop strategies to enhance customer retention and capitalize on upsell opportunities.

   In the SaaS industry, CLV is often calculated based on factors such as monthly recurring revenue (MRR), customer acquisition costs, churn rates, and upgrade or downgrade rates. By analyzing these metrics, SaaS companies can identify the optimal pricing strategies, customer acquisition channels, and retention initiatives to maximize the lifetime value of their customer base.

   SaaS companies can leverage CLV analysis to segment their customers based on their potential value and tailor their offerings and support accordingly. For example, high-CLV customers may be offered premium support, additional features, or personalized onboarding to ensure their long-term satisfaction and retention. Additionally, SaaS businesses can use CLV data to optimize their product roadmaps and development efforts. By understanding the features and functionalities that drive customer retention and upsell opportunities, they can prioritize product enhancements that cater to the needs of their most valuable customers.

5. Retail

   In retail, CLV helps businesses understand the purchasing behavior of their customers and develop targeted marketing campaigns. By focusing on high-CLV customers, retailers can increase customer loyalty and drive long-term revenue growth. Retail CLV analysis often involves tracking customer purchase history, basket sizes, and engagement levels across various channels, including brick-and-mortar stores, e-commerce platforms, and loyalty programs. This data can be used to segment customers based on their spending patterns, preferences, and lifetime value.

   Retailers can leverage CLV data to develop targeted loyalty programs, personalized promotions, and tailored marketing campaigns. For example, high-CLV customers may receive exclusive offers, early access to new products, or personalized recommendations based on their purchase history. Additionally, CLV analysis can help retailers optimize their inventory management and merchandising strategies. By understanding the preferences and purchasing behavior of their most valuable customers, retailers can ensure that they stock the right products and allocate shelf space effectively to maximize sales and profitability.

Challenges and Considerations in Implementing CLV

While CLV is a powerful metric for businesses, its implementation and effective utilization can present several challenges:

1. Data Quality and Availability

   Accurate CLV calculations rely on high-quality data, including customer purchase history, demographic information, and engagement metrics. Businesses may face challenges in collecting and integrating data from multiple sources, such as point-of-sale systems, e-commerce platforms, and customer relationship management (CRM) systems.

Incomplete or inaccurate data can lead to flawed CLV calculations and potentially misleading insights. To address this challenge, businesses must invest in robust data management systems and implement processes to ensure data quality and consistency across various touchpoints.

2. Changing Customer Behavior

   Customer behavior is dynamic and can change over time due to various factors, including market trends, economic conditions, and personal circumstances. CLV models that rely solely on historical data may fail to capture these shifts, leading to inaccurate predictions and suboptimal decision-making.

   To mitigate this challenge, businesses should continuously monitor customer behavior and update their CLV models accordingly. This may involve incorporating external data sources, such as economic indicators or market research, to account for broader trends and factors that may influence customer behavior.

3. Accounting for Customer Acquisition Costs

   While CLV calculations typically focus on revenue streams, it is essential to consider customer acquisition costs (CAC) to accurately assess profitability. Failing to account for CAC can lead to an overestimation of CLV, particularly in industries with high customer acquisition costs.

   Businesses should carefully track and allocate customer acquisition costs, including marketing expenses, sales commissions, and other related expenses. By incorporating CAC into their CLV calculations, businesses can gain a more comprehensive understanding of the true profitability of their customer relationships.

4. Balancing Short-term and Long-term Strategies

   CLV is a long-term metric that focuses on the lifetime value of customers. However, businesses must also consider short-term objectives, such as meeting quarterly or annual revenue

targets. Striking the right balance between short-term and long-term strategies can be challenging, as investments in customer retention and loyalty may not yield immediate returns.
To address this challenge, businesses should develop a holistic approach that aligns short-term tactics with long-term CLV objectives. This may involve setting appropriate Key Performance Indicators (KPIs) that balance immediate revenue goals with customer lifetime value metrics, and continuously monitoring and adjusting strategies as needed.

5. Organizational Alignment and Buy-in

   Implementing an effective CLV strategy requires cross-functional collaboration and buy-in from various stakeholders within the organization. Marketing, sales, customer service, and product teams must work together to ensure a consistent and cohesive approach to customer acquisition, retention, and value maximization.

   Achieving organizational alignment can be challenging, as different departments may have competing priorities or siloed approaches. To overcome this challenge, businesses should foster a customer-centric culture, establish clear communication channels, and provide training and education to ensure a shared understanding of CLV and its importance across the organization.

Best Practices for Leveraging CLV

To effectively leverage CLV and drive business success, businesses should consider the following best practices:

1. Integrate CLV into Decision-Making Processes

   CLV should be a key consideration in various business decisions, including marketing campaigns, product development, pricing strategies, and resource allocation. By integrating CLV into decision-making processes, businesses can ensure that their actions are aligned with maximizing customer lifetime value and long-term profitability.

2. Continuously Monitor and Refine CLV Models

   Customer behavior and market conditions are constantly evolving, necessitating regular monitoring and refinement of CLV models. Businesses should establish processes to review and update their CLV calculations periodically, incorporating new data sources, changing customer preferences, and emerging trends.

3. Invest in Customer Data Management

   Accurate CLV calculations rely on high-quality customer data. Businesses should invest in robust data management systems and processes to ensure data integrity, consistency, and accessibility across various touchpoints. This may involve implementing customer relationship management (CRM) systems, data warehousing solutions, and data governance frameworks.

4. Foster a Customer-Centric Culture

   Maximizing CLV requires a customer-centric mindset throughout the organization. Businesses should cultivate a culture that prioritizes customer satisfaction, loyalty, and long-term value creation. This can be achieved through employee training, incentive programs, and clear communication of the organization's customer-centric vision and values.

5. Leverage Advanced Analytics and Predictive Modeling

   As CLV models become more sophisticated, businesses should leverage advanced analytics techniques and predictive modeling to enhance their insights and decision-making capabilities. This may involve utilizing machine learning algorithms, artificial intelligence, and predictive analytics tools to identify patterns, forecast customer behavior, and optimize strategies for maximizing CLV.

6. Collaborate with Cross-Functional Teams

   Effective CLV management requires collaboration across various departments, including marketing, sales, customer service, and product development. Businesses should establish

cross-functional teams or working groups to share insights, align strategies, and ensure a cohesive approach to customer acquisition, retention, and value maximization.

7. Measure and Optimize Customer Experiences

   Understanding CLV helps businesses identify the factors that contribute to a positive customer experience. By focusing on these factors, companies can enhance the overall customer journey, leading to higher satisfaction and loyalty. Personalized experiences, timely customer support, and relevant product recommendations are some ways to improve CLV.

Types of CLV Models

Customer Lifetime Value (CLV) is a crucial metric for businesses, as it measures the total revenue a company can expect from a single customer account throughout the entire duration of their relationship. Accurately calculating and leveraging CLV can significantly impact a company's profitability, customer retention, and overall business strategy. There are several types of CLV models available, each with its own approach, strengths, and weaknesses. In this comprehensive response, we delve into the different types of CLV models, their formulas, and how they are calculated, providing a detailed understanding of this vital metric.

Simple CLV Model

The Simple CLV model is the most basic approach to calculating CLV. It is straightforward and easy to implement, making it a good starting point for businesses with limited data or resources. However, it comes with significant limitations in terms of accuracy and predictive power.

Formula:

CLV = Average Customer Revenue × Average Customer Lifespan

Explanation:

- **Average Customer Revenue:** This is the average amount of revenue generated by a customer over a specific period, usually a year. It is calculated by dividing the total revenue from all customers by the number of customers.
- **Average Customer Lifespan:** This is the average duration a customer remains active with the business. It is calculated by taking the sum of all customer lifetimes and dividing it by the total number of customers.

Calculation steps:

1. Determine the time period for which you want to calculate the average customer revenue (e.g., one year).
2. Calculate the total revenue generated by all customers during that time period.
3. Divide the total revenue by the number of customers to get the average customer revenue.
4. Calculate the average customer lifespan by summing the lifetimes of all customers and dividing by the total number of customers.
5. Multiply the average customer revenue by the average customer lifespan to get the Simple CLV.

Advantages:

- Easy to calculate and understand.
- Requires minimal data.
- Provides a quick estimate of customer value.

Disadvantages:

- Assumes constant revenue and churn rates, which is rarely the case in real-world scenarios.
- Does not account for the time value of money or changes in customer behavior over time.

- Does not consider the costs associated with acquiring and retaining customers.
- Provides a less accurate estimate of CLV compared to more advanced models.

While the Simple CLV model is a good starting point, it is often too simplistic for businesses that require a more accurate and comprehensive understanding of customer value. More advanced models are needed to account for the complexities of customer behavior and the time value of money.

Historical CLV Model

The Historical CLV model calculates CLV based on past customer data. It sums all the historical revenues generated by a customer without attempting to predict future behavior. This model is useful for businesses that want to understand the value of their existing customer base without making assumptions about future behavior.

Formula:

$CLV = \sum(t=0 \text{ to } T) \text{ Revenue}(t)$

Explanation:

- **Revenue(t):** The revenue generated by the customer at time *t*.
- **T:** The total time period considered for the calculation.

Calculation steps:

1. Determine the time period for which you want to calculate the historical CLV (e.g., the lifetime of the customer or a specific number of years).
2. For each customer, sum the revenue generated at each time period t within the specified time frame.
3. The resulting sum is the Historical CLV for that customer.

Advantages:

- Simple and straightforward.
- Based on actual historical data, making it relatively accurate for past performance.
- Useful for understanding the value of existing customers.

Disadvantages:

- Does not predict future customer behavior.
- Ignores potential future revenue and changes in customer behavior.
- Not useful for businesses looking to forecast future revenue streams.
- Does not account for the time value of money or the costs associated with acquiring and retaining customers.

While the Historical CLV model provides valuable insights into the value of existing customers, it does not offer a complete picture of customer lifetime value. Businesses that want to forecast future revenue and optimize their customer acquisition and retention strategies need to consider more advanced predictive models.

Predictive CLV Models

Predictive CLV models use statistical techniques and machine learning to forecast future customer behavior and estimate CLV more accurately. These models are more complex but provide a more comprehensive understanding of customer value, making them essential for businesses that want to optimize their customer strategies and maximize profitability.

a. Regression Models
   Regression models use various customer attributes and transaction history to predict future customer behavior and estimate CLV.
   Formula:

$$CLV = \beta 0 + \beta 1\ X1 + \beta 2\ X2 + \ldots + \beta n\ Xn$$

Explanation:

- **β0:** Intercept term.
- **βi:** Coefficients for each predictor variable Xi.
- **Xi:** Predictor variables such as customer demographics, purchase history, and engagement metrics.

Calculation steps:

1. Collect data on customer attributes and transaction history, including predictor variables such as demographics, purchase behavior, and engagement metrics.
2. Split the data into training and testing sets.
3. Use the training data to fit a regression model, estimating the coefficients βi for each predictor variable.
4. Evaluate the model's performance on the testing data and make any necessary adjustments.
5. For each customer, calculate the CLV by substituting their predictor variable values into the regression equation.

Advantages:

- Can incorporate multiple variables to improve accuracy.
- Flexible and can be adapted to different business contexts.
- Provides insights into the factors influencing customer behavior and CLV.

Disadvantages:

- Requires a significant amount of data for training.
- Can be complex to implement and interpret.
- Assumes a linear relationship between predictor variables and CLV, which may not always be the case.

Example Regression Model:
Suppose a company wants to predict CLV based on a customer's age, average order value, and frequency of purchases. The regression model might look like:

CLV = 100 + 2.5 × Age + 50 × Average Order Value + 75 × Purchase Frequency

In this example, the model suggests that older customers, those with higher average order values, and those who purchase more frequently tend to have higher CLVs.

b. Probabilistic Models

Probabilistic models use probability distributions to model customer behavior. One of the most well-known probabilistic models is the Pareto/NBD (Negative Binomial Distribution) model.

Pareto/NBD Model:

This model assumes that customer purchases follow a Poisson process and that the dropout rate follows a Gamma distribution.

Formula:

CLV = (p × λ) / μ

Explanation:

- **p:** Probability of a customer being active.
- **λ:** Purchase rate.
- **μ:** Dropout rate.

Calculation steps:

1. Collect data on customer purchase history and dropout behavior.
2. Estimate the parameters p, λ, and μ using maximum likelihood estimation or other techniques.

3. Substitute the estimated parameter values into the Pareto/NBD formula to calculate the CLV for each customer.

Advantages:

– Provides a probabilistic estimate of customer behavior.

– Can handle varying purchase frequencies and dropout rates.

– Accounts for the heterogeneity in customer behavior.

Disadvantages:

– Requires a good understanding of probability distributions.

– Can be computationally intensive.

– May not capture all aspects of customer behavior, such as the impact of marketing campaigns or changes in customer preferences.

Example Pareto/NBD Model:

Suppose a company has the following estimated parameters for its customer base:

– $p = 0.8$: The probability that a randomly selected customer is still active.

– $\lambda = 2$: The average number of purchases per year for an active customer.

– $\mu = 0.2$: The d opout rate, or the probability that a customer becomes inactive in a given year.

Using the Pareto/NBD formula, the CLV for this customer base would be:

$CLV = (0.8 \times 2) / 0.2 = 8$

This suggests that, on average, each customer is expected to generate a lifetime value of 8 times the average purchase amount.

c. Machine Learning Models

Machine learning models, such as decision trees, random forests, and neural networks, can be used to predict CLV. These models can handle large datasets and complex relationships between variables, making them well-suited for businesses with rich customer data.

Calculation steps:

1. Collect data on customer attributes, transaction history, and other relevant variables.
2. Split the data into training and testing sets.
3. Choose an appropriate machine learning algorithm (e.g., random forest, neural network) and train the model on the training data.
4. Evaluate the model's performance on the testing data and make any necessary adjustments.
5. Use the trained model to predict the CLV for each customer based on their input variables.

Advantages:

– Can capture non-linear relationships and interactions between variables.
– Highly accurate with sufficient data and proper tuning.
– Can handle large and complex datasets.

Disadvantages:

– Requires significant computational resources.
– Can be a “black box,” making it difficult to interpret the results.
– Prone to overfitting if not properly regularized or validated.

Example Machine Learning Model:

Suppose a company wants to predict CLV using a random forest model. The model might consider variables such as:

- Customer demographics (age, gender, location)
- Purchase history (frequency, average order value, product categories)
- Engagement metrics (website visits, email opens, social media interactions)
- Marketing campaign responses

The random forest model would learn patterns and relationships between these variables and CLV from the training data, allowing it to make accurate predictions for new customers or existing customers with changing behavior.

Survival Analysis Models

Survival analysis models, also known as "time-to-event" models, are particularly useful for subscription-based businesses. These models estimate the probability of a customer remaining active (surviving) over time, taking into account censored data (customers who are still active at the time of analysis).

Survival Function:

The survival function, (S(t)), represents the probability that a customer will remain active beyond time (t):

$$S(t) = P(T > t)$$

Where (T) is the time to churn. This function is decreasing over time, starting at 1 when (t = 0) and approaching 0 as (t) approaches infinity.

Hazard Function:

The hazard function, lambda(t), is the instantaneous rate of churn at time *t*, given that the customer has survived up to time *t*:

$$\lambda(t) = \lim_{\Delta t \to 0} \frac{P(t \le T < t + \Delta t | T \ge t)}{\Delta t}$$

The hazard function provides insights into the risk of churn at any given time.

Relationship Between Survival and Hazard Functions:

The survival and hazard functions are related by the following equations:

$$S(t) = \exp\left(-\int_0^t \lambda(u)du \right)$$

$$\lambda(t) = -\frac{d}{dt}\ln\left(S(t)\right)$$

Common Survival Analysis Models:

- **Cox Proportional Hazards model:** A semi-parametric model that estimates the hazard (risk) of an event occurring at a particular time.

$$\lambda(t|X) = \lambda_0(t)\exp(\beta' X)$$

Where:

- lambda_0(t) is the baseline hazard function.
- beta is the vector of coefficients for covariates X.

The partial likelihood for the Cox model is:

$$L(\beta) = \prod_{i=1}^{D} \frac{\exp(\beta' X_i)}{\sum_{j \in R_i} \exp(\beta' X_j)}$$

Where:

D is the set of individuals who have experienced the event.

- R_i is the risk set at time t_i.

- **Kaplan-Meier estimator:** A non-parametric statistic used to estimate the survival function from lifetime data.

$$\hat{S}(t) = \prod_{t_i \le t} \left(1 - \frac{d_i}{n_i} \right)$$

Where:

- t_i is the time of the ith event.
- d_i is the number of events (churns) at time t_i.
- n_i is the number of customers at risk just before time t_i.

Example calculation:
Assume we have the following churn data:
The Kaplan-Meier survival function is calculated as follows:

$$\hat{S}(1) = 1 - \frac{1}{10} = 0.9$$
$$\hat{S}(2) = 0.9 \times \left(1 - \frac{2}{9}\right) = 0.7$$
$$\hat{S}(3) = 0.7 \times \left(1 - \frac{1}{7}\right) = 0.6$$

- **Weibull model:** A parametric model that assumes the time-to-event follows a Weibull distribution.

Calculation steps:

1. Collect data on customer lifetimes, censoring indicators, and relevant covariates (e.g., demographics, usage patterns, and marketing campaigns).
2. Fit the survival analysis model (e.g., Cox Proportional Hazards model) to the data, estimating the coefficients βi for each covariate.
3. Use the fitted model to estimate the survival probability for each customer at different points in time.
4. Calculate the CLV by integrating the survival probability over time, considering the expected revenue and costs associated with each customer.

Advantages:

– Can handle censored data, making it suitable for ongoing customer relationships.
– Provides insights into the factors influencing customer survival.
– Can incorporate time-varying covariates.
– Suitable for businesses with recurring revenue streams.

Disadvantages:

- Requires specialized knowledge to implement and interpret.
- Can be computationally intensive.
- Assumes proportional hazards or a specific distribution for the time-to-event, which may not always hold.

Example Survival Analysis Model:

Suppose a subscription-based company wants to estimate CLV using the Cox Proportional Hazards model. The model might include covariates such as:

- Customer tenure
- Subscription plan
- Usage patterns (e.g., number of logins, feature usage)
- Marketing campaign exposure

The model would estimate the hazard of a customer churning (becoming inactive) at any given time, based on these covariates. The survival probability over time can then be used to calculate the expected future revenue and, ultimately, the CLV for each customer.

Traditional CLV Model

The Traditional CLV model is suitable for businesses where customer revenues do not remain constant over time. It factors in changes that occur across the customer lifetime, such as varying revenue streams and the costs associated with retaining customers.

Formula:

CLV = (GML × Retention Rate) / (1 + Discount Rate - Retention Rate)

Explanation:

- **GML:** Gross margin per customer lifespan.
- **Retention Rate:** The percentage of customers who stay with the business over a set time period.
- **Discount Rate:** A percentage to account for inflation, often set at 10%.

Calculation steps:

1. Determine the gross margin per customer lifespan (GML) by subtracting the costs associated with acquiring and serving a customer from the total revenue generated by that customer over their lifetime.
2. Calculate the retention rate by dividing the number of customers retained over a specific time period by the total number of customers at the beginning of that period.
3. Choose an appropriate discount rate to account for the time value of money and inflation.
4. Substitute the calculated values into the Traditional CLV formula.

   Advantages:

   – Considers the time value of money and the costs associated with retaining customers.
   – Accounts for changes in customer revenue over time.
   – Provides a more comprehensive estimate of CLV compared to the Simple CLV model.

   Disadvantages:

   – Assumes a constant retention rate and discount rate over time, which may not always be accurate.
   – Does not account for individual customer characteristics or behavior.
   – Requires accurate estimates of gross margins and retention rates, which can be challenging to obtain.

   Example Traditional CLV Model:

   Suppose a company has the following data:

   – Gross margin per customer lifespan (GML) = $1,000
   – Retention rate = 80% (i.e., 80% of customers are retained each year)
   – Discount rate = 10%

Using the Traditional CLV formula, the CLV would be:

$CLV = (1{,}000 \times 0.8) / (1 + 0.1 - 0.8) = 800 / 0.3 = \$2{,}666.67$

This suggests that, on average, each customer is expected to generate a lifetime value of $2,666.67 for the company, considering the time value of money and the costs associated with retaining customers.

Accelerated Failure Time (AFT) Model

The Accelerated Failure Time (AFT) model is a parametric model used in survival analysis to estimate the effect of covariates on the survival time of subjects. Unlike the Cox Proportional Hazards model, which assumes that covariates multiplicatively affect the hazard function, the AFT model assumes that covariates accelerate or decelerate the life course of an event by a constant factor. This makes the AFT model particularly useful in scenarios where the proportional hazards assumption may not hold.

Model Specification:

In the AFT model, the logarithm of the survival time is modeled as a linear function of the covariates. The general form of the AFT model can be specified as:

$\log(T) = \beta 0 + \beta 1\ X1 + \beta 2\ X2 + \ldots + \beta p\ Xp + \varepsilon$

Where:

- T is the survival time.
- $\beta 0$ is the intercept.
- βi are the coefficients for the covariates Xi.
- ε is the error term, which follows a specific distribution (e.g., normal, logistic, Weibull).

The model can also be expressed in terms of the survival function $S(t|\theta)$ and the hazard function $\lambda(t|\theta)$:

$S(t|\theta) = S0(\theta\ t)$
$\lambda(t|\theta) = \theta\ \lambda 0(\theta\ t)$

Where $\theta = \exp(-[\beta 1\ X1 + \ldots + \beta p\ Xp])$.

Advantages of AFT models:

1. No proportional hazards assumption: Unlike the Cox PH model, AFT models do not require the proportional hazards assumption, which states that the hazard ratio between different levels of a covariate is constant over time. This assumption is often violated in real-world data.
2. Direct interpretation: The coefficients in an AFT model directly quantify the effect of covariates on the survival time, making the interpretation more intuitive. The coefficients represent the acceleration or deceleration of the survival time.
3. More efficient estimates: Based on asymptotic results, AFT models can lead to more efficient parameter estimates than the Cox PH model, especially when the sample size is small.
4. Insights into baseline hazard: When sufficient empirical information is available, AFT models can provide insights into the form of the baseline hazard function, which is not possible with the semi-parametric Cox PH model.
5. Handling of censored data: Like the Cox PH model, AFT models can handle censored survival data.
6. Collapsibility: AFT models are collapsible, meaning that the marginal and conditional estimates of the coefficients are the same. This property is useful for causal inference.

Disadvantages of AFT models:

1. Distributional assumption: AFT models require a specific distributional assumption for the survival times (e.g., Weibull, log-logistic, log-normal), which may not always hold in practice.
2. Sensitivity to outliers: AFT models can be sensitive to outliers or extreme values in the data, which can affect the parameter estimates.
3. Difficulty in model selection: Choosing the appropriate distribution for the AFT model can be challenging, and different distributions may lead to different results.

4. Interpretation for time-varying effects: While AFT models can handle time-varying covariates, the interpretation of their effects can be more complex compared to the Cox PH model.
5. Limited software availability: Some statistical software packages may have limited support or implementation for AFT models compared to the widely available Cox PH model.

Example 11.1

This example creates synthetic data to generate RFM segments to show how the concept of RFM segments works using PySpark:

```
# Import necessary libraries
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, sum, max, lit, datediff,
when, concat, percentile_approx
from pyspark.sql.types import DateType
import datetime
import random

# Initialize Spark session
spark = SparkSession.builder.appName("RFM Analysis").getOrCreate()

# Generate synthetic data
num_customers = 100
num_transactions = 1000
customers = [f"customer_{i}" for i in range(1, num_customers + 1)]
transaction_dates = [datetime.date.today() - datetime.
timedelta(days=random.randint(1, 365)) for _ in range(num_transactions)]
transaction_amounts = [random.uniform(10.0, 1000.0) for _ in range(num_
transactions)]

data = [(random.choice(customers), transaction_dates[i], transaction_
amounts[i]) for i in range(num_transactions)]
columns = ["CustomerID", "TransactionDate", "Amount"]

# Create DataFrame
df = spark.createDataFrame(data, columns)
```

```
# Show the data
df.show(5)

# Calculate Recency, Frequency, and Monetary values
current_date = datetime.date.today()

# Add a column with the current date
df = df.withColumn("CurrentDate", lit(current_date).cast(DateType()))

# Calculate Recency
recency_df = df.groupBy("CustomerID").agg(datediff(max("CurrentDate"),
max("TransactionDate")).alias("Recency"))

# Calculate Frequency
frequency_df = df.groupBy("CustomerID").agg(count("TransactionDate").
alias("Frequency"))

# Calculate Monetary
monetary_df = df.groupBy("CustomerID").agg(sum("Amount").alias("Monetary"))

# Combine Recency, Frequency, and Monetary values
rfm_df = recency_df.join(frequency_df, "CustomerID").join(monetary_df,
"CustomerID")

# Show the RFM values
rfm_df.show(5)

# Calculate dynamic quantiles
quantiles = [0.25, 0.50, 0.75]

recency_quantiles = rfm_df.approxQuantile("Recency", quantiles, 0.0)
frequency_quantiles = rfm_df.approxQuantile("Frequency", quantiles, 0.0)
monetary_quantiles = rfm_df.approxQuantile("Monetary", quantiles, 0.0)

# Define segment conditions using quantiles
rfm_df = rfm_df.withColumn("R_Segment",
    when(col("Recency") <= recency_quantiles[0], "R1")
    .when(col("Recency") <= recency_quantiles[1], "R2")
    .when(col("Recency") <= recency_quantiles[2], "R3")
    .otherwise("R4"))
```

```
rfm_df = rfm_df.withColumn("F_Segment",
    when(col("Frequency") > frequency_quantiles[2], "F1")
    .when(col("Frequency") > frequency_quantiles[1], "F2")
    .when(col("Frequency") > frequency_quantiles[0], "F3")
    .otherwise("F4"))

rfm_df = rfm_df.withColumn("M_Segment",
    when(col("Monetary") > monetary_quantiles[2], "M1")
    .when(col("Monetary") > monetary_quantiles[1], "M2")
    .when(col("Monetary") > monetary_quantiles[0], "M3")
    .otherwise("M4"))

rfm_df = rfm_df.withColumn("RFM_Segment", concat(col("R_Segment"),
col("F_Segment"), col("M_Segment")))

# Show the segmented RFM values
rfm_df.show(10)

# Stop the Spark session
spark.stop()
```

Example 11.2

We again use synthetic data to generate CLV values using AFT libraries from PySpark:

```
# Import necessary libraries
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, sum, max, lit, datediff,
when, min, udf
from pyspark.sql.types import DateType, FloatType
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import AFTSurvivalRegression
from pyspark.ml.evaluation import RegressionEvaluator
import datetime
import random

# Initialize Spark session
spark = SparkSession.builder.appName("Survival Analysis for CLV with AFT").
getOrCreate()
```

```
# Generate synthetic data
num_customers = 100
num_transactions = 1000
customers = [f"customer_{i}" for i in range(1, num_customers + 1)]
transaction_dates = [datetime.date.today() - datetime.
timedelta(days=random.randint(1, 365)) for _ in range(num_transactions)]
transaction_amounts = [random.uniform(10.0, 1000.0) for _ in range(num_
transactions)]
churn_dates = [random.choice([datetime.date.today() - datetime.
timedelta(days=random.randint(30, 365)), None]) for _ in range(num_
customers)]

transaction_data = [(random.choice(customers), transaction_dates[i],
transaction_amounts[i]) for i in range(num_transactions)]
churn_data = [(customers[i], churn_dates[i]) for i in range(num_customers)]

transaction_columns = ["CustomerID", "TransactionDate", "Amount"]
churn_columns = ["CustomerID", "ChurnDate"]

# Create DataFrames
transaction_df = spark.createDataFrame(transaction_data, transaction_
columns)
churn_df = spark.createDataFrame(churn_data, churn_columns)

# Show the data
transaction_df.show(5)
churn_df.show(5)

# Calculate Recency, Frequency, and Monetary values
current_date = datetime.date.today()

# Add a column with the current date
transaction_df = transaction_df.withColumn("CurrentDate", lit(current_
date).cast(DateType()))

# Calculate Recency
recency_df = transaction_df.groupBy("CustomerID").agg(datediff(max("Current
Date"), max("TransactionDate")).alias("Recency"))
```

```
# Calculate Frequency
frequency_df = transaction_df.groupBy("CustomerID").
agg(count("TransactionDate").alias("Frequency"))

# Calculate Monetary
monetary_df = transaction_df.groupBy("CustomerID").agg(sum("Amount").
alias("Monetary"))

# Combine Recency, Frequency, and Monetary values
rfm_df = recency_df.join(frequency_df, "CustomerID").join(monetary_df,
"CustomerID")

# Join RFM data with churn data
rfm_churn_df = rfm_df.join(churn_df, "CustomerID")

# Fill null ChurnDate with a future date to indicate no churn
future_date = current_date + datetime.timedelta(days=365)
rfm_churn_df = rfm_churn_df.withColumn("ChurnDate", when(col("ChurnDate").
isNull(), lit(future_date)).otherwise(col("ChurnDate")))

# Calculate customer tenure (days between first transaction and churn date)
tenure_df = transaction_df.groupBy("CustomerID").
agg(min("TransactionDate").alias("FirstTransactionDate"))
rfm_churn_df = rfm_churn_df.join(tenure_df, "CustomerID")
rfm_churn_df = rfm_churn_df.withColumn("Tenure", datediff(col("ChurnDate"),
col("FirstTransactionDate")))

# Calculate event (churned or not)
rfm_churn_df = rfm_churn_df.withColumn("Event", when(col("ChurnDate") <
future_date, 1).otherwise(0))

# Filter out rows with zero or negative tenure (required for AFT model)
rfm_churn_df = rfm_churn_df.filter(col("Tenure") > 0)

# Assemble features for AFT model
assembler = VectorAssembler(inputCols=["Recency", "Frequency", "Monetary"],
outputCol="features")
rfm_churn_df = assembler.transform(rfm_churn_df)
```

```
# Split the data into training and test sets
train_df, test_df = rfm_churn_df.randomSplit([0.8, 0.2], seed=12345)

# Fit the AFT Survival Regression model
aft = AFTSurvivalRegression(featuresCol="features", labelCol="Tenure",
censorCol="Event")
aft_model = aft.fit(train_df)

# Predict survival probabilities for the test set
predictions = aft_model.transform(test_df)

# Show predictions
predictions.select("CustomerID", "features", "Event", "Tenure",
"prediction").show(5)

# Calculate average monthly revenue
rfm_churn_df = rfm_churn_df.withColumn("AvgMonthlyRevenue", col("Monetary")
/ (col("Tenure") / 30))

# Function to calculate CLV using survival analysis
def calculate_clv(rfm_churn_df, predictions):
    # Join predictions with original dataframe
    rfm_churn_df = rfm_churn_df.join(predictions.select("CustomerID",
    "prediction"), "CustomerID")

    # Calculate CLV
    rfm_churn_df = rfm_churn_df.withColumn("PredictedCLV",
    col("AvgMonthlyRevenue") * col("prediction") / 30)
    # Convert days to months
    return rfm_churn_df

# Apply the CLV calculation
clv_df = calculate_clv(rfm_churn_df, predictions)

# Calculate actual CLV using actual churn date
clv_df = clv_df.withColumn("ActualCLV", col("AvgMonthlyRevenue") *
(col("Tenure") / 30))
```

```
# Ensure that the necessary columns are available for evaluation
clv_df_eval = clv_df.select("PredictedCLV", "ActualCLV")

# Show the final CLV data
clv_df.select("CustomerID", "Recency", "Frequency", "Monetary", "Tenure",
"AvgMonthlyRevenue", "prediction", "PredictedCLV", "ActualCLV").show(10)

# Evaluate the accuracy of CLV predictions
evaluator_mae = RegressionEvaluator(labelCol="ActualCLV",
predictionCol="PredictedCLV", metricName="mae")
mae = evaluator_mae.evaluate(clv_df_eval)
print(f"Mean Absolute Error (MAE): {mae}")

evaluator_mse = RegressionEvaluator(labelCol="ActualCLV",
predictionCol="PredictedCLV", metricName="mse")
mse = evaluator_mse.evaluate(clv_df_eval)
print(f"Mean Squared Error (MSE): {mse}")

evaluator_rmse = RegressionEvaluator(labelCol="ActualCLV",
predictionCol="PredictedCLV", metricName="rmse")
rmse = evaluator_rmse.evaluate(clv_df_eval)
print(f"Root Mean Squared Error (RMSE): {rmse}")

# Stop the Spark session
spark.stop()
```

Return on Marketing Investment (ROMI)

Return on Marketing Investment (ROMI) is a critical metric for businesses to measure the effectiveness of their marketing campaigns. It quantifies the revenue generated by marketing activities relative to the costs incurred, helping businesses optimize their marketing spend. In today's data-driven world, calculating ROMI accurately involves leveraging multiple data science models and techniques to capture the complexities of consumer behavior and marketing impacts.

The Basic Formula for ROMI

The basic formula for ROMI is:

$$\text{ROMI} = \frac{\text{Incremental Revenue} - \text{Marketing Cost}}{\text{Marketing Cost}} \times 100$$

Where:

- **Incremental Revenue**: The additional revenue generated by the marketing campaign.
- **Marketing Cost**: The total cost of the marketing campaign.

This formula provides a high-level view of ROMI, but in practice, the calculation is more complex and involves various data science techniques to ensure accuracy and reliability.

Step-by-Step Approach to Calculating ROMI Using Data Science Models

Step 1: Data Collection and Preparation

Data collection is the first step in calculating ROMI. It involves gathering data from various sources to provide a comprehensive view of marketing activities and their impact on revenue. This includes:

- **Customer Data:** Demographic information, purchase history, and behavior data.
- **Marketing Data:** Details of marketing campaigns, including costs, impressions, clicks, and conversions.
- **Sales Data:** Records of transactions, order values, and dates.
- **Web Analytics Data:** Website traffic, user interactions, and conversion rates.

Each data source provides valuable insights that contribute to a holistic understanding of marketing effectiveness.

Data preparation involves cleaning, transforming, and merging these data sources to create a unified dataset suitable for analysis.

- **Data Cleaning:** Remove duplicates, handle missing values, and correct errors to ensure data quality.
- **Data Transformation:** Convert data into a suitable format for analysis. This may involve aggregating daily data into monthly data, normalizing numerical values, or encoding categorical variables.
- **Data Merging:** Combine data from different sources to create a single dataset that includes all relevant information.

Example: An e-commerce retailer collects data from its CRM system, marketing automation platform, e-commerce website, and web analytics tools. The data is cleaned, transformed, and merged to create a comprehensive dataset for analysis.

Step 2: Customer Segmentation

Customer segmentation involves dividing customers into distinct groups based on common characteristics. This helps tailor marketing strategies to different customer segments and determine how various segments respond to marketing efforts.

Techniques for customer segmentation:

- **K-Means Clustering:** A popular algorithm that partitions customers into K clusters based on their similarity.
- **Hierarchical Clustering:** Builds a hierarchy of clusters by merging or splitting existing clusters.
- DBSCAN (Density-Based Spatial Clustering of Applications with Noise): Identifies clusters of varying shapes and sizes based on density.

Purpose of customer segmentation:

- Tailor marketing strategies to different customer segments.
- Analyze the impact of marketing campaigns on various customer groups.
- Identify high-value segments to focus marketing efforts.

Example: Using K-Means clustering, an e-commerce retailer segments customers based on Recency, Frequency, and Monetary (RFM) values. This helps the retailer understand which segments are more likely to respond to specific marketing campaigns and allocate budget accordingly.

Step 3: Attribution Modeling

Attribution modeling determines which marketing channels and touchpoints contributed to a conversion. Accurate attribution is essential for understanding the true impact of marketing activities and optimizing future campaigns.

Types of attribution models:

1. **Single-Touch Attribution:**
   - **First-Touch Attribution**: Attributes the entire conversion value to the first touchpoint.
   - **Last-Touch Attribution**: Attributes the entire conversion value to the last touchpoint.
2. **Multi-Touch Attribution:**
   - **Linear Attribution**: Distributes the conversion value equally across all touchpoints.
   - **Time-Decay Attribution**: Attributes more value to touchpoints closer to the conversion.
   - **Position-Based Attribution**: Assigns a higher value to the first and last touchpoints, with the remaining value distributed among the middle touchpoints.
3. **Algorithmic Attribution**: Uses machine learning models to assign conversion value based on the actual impact of each touchpoint. This method is more sophisticated and can provide more accurate insights compared to rule-based models.

Purpose of attribution modeling:

- Understand the contribution of each marketing channel to conversions.
- Allocate marketing budget more effectively.
- Identify which touchpoints and channels are most effective in driving conversions.

Example: An e-commerce retailer uses a linear attribution model to distribute the conversion value equally across all touchpoints. By doing so, the retailer gains insights into how each marketing channel (e.g., email, social media, search engine) contributes to the overall conversion, allowing for more informed budget allocation decisions.

Step 4: Predictive Modeling for Incremental Revenue

Predictive modeling uses historical data to forecast the revenue impact of marketing campaigns. This involves applying regression analysis and machine learning algorithms to predict future outcomes based on past behavior.

Common predictive models:

- **Linear Regression:** Models the relationship between dependent (e.g., revenue) and independent variables (e.g., marketing spend).
- **Time Series Analysis:** Techniques such as ARIMA (AutoRegressive Integrated Moving Average) and exponential smoothing forecast future values based on past data.
- **Machine Learning Models:** Algorithms such as decision trees, random forests, gradient boosting, and neural networks capture complex relationships and make accurate predictions.

Purpose of predictive modeling:

- Forecast revenue impact of future marketing campaigns.
- Optimize marketing spend based on predicted outcomes.
- Identify key drivers of revenue and focus marketing efforts on high-impact activities.

Example: An e-commerce retailer uses a Gradient Boosting Regressor to predict the incremental revenue generated by a marketing campaign. The model is trained using historical data, and its performance is evaluated using metrics such as mean squared error. This helps the retailer forecast the potential revenue impact of future campaigns and make data-driven decisions about marketing spend.

Step 5: Customer Lifetime Value (CLV) Analysis

Customer Lifetime Value (CLV) is the total revenue a business can expect from a customer over their lifetime. It is a crucial metric for understanding the long-term impact of marketing efforts and tailoring strategies to maximize customer value.

Purpose of CLV analysis:

- Understand the long-term value of customers acquired through marketing campaigns.
- Tailor marketing strategies to maximize CLV.
- Identify high-value customers and focus retention efforts on them.

Example: An e-commerce retailer calculates historical and predictive CLV to understand the long-term impact of a marketing campaign. Predictive models estimate future CLV based on customer characteristics and behavior, helping the retailer tailor marketing strategies to maximize customer value.

Step 6: Marketing Cost Calculation

Marketing cost includes all expenses associated with running a marketing campaign. This encompasses direct costs such as advertising spend and indirect costs such as personnel and technology.

Components of marketing cost:

- **Advertising Spend:** Costs related to online and offline advertising channels.
- **Personnel Costs:** Salaries and benefits of marketing staff.
- **Technology Costs:** Costs of marketing automation platforms, analytics tools, and other technologies.
- **Creative Costs:** Costs associated with creating marketing content, such as graphic design and video production.

Purpose of marketing cost calculation:

- Accurately account for all marketing expenses.
- Calculate the net profit generated by marketing activities.
- Ensure a comprehensive understanding of the total investment in marketing campaigns.

Example: An e-commerce retailer summarizes all marketing costs, including advertising spend, personnel costs, technology costs, and creative costs. By accurately accounting for these expenses, the retailer ensures that the ROMI calculation reflects the true investment in the marketing campaign.

Step 7: ROMI Calculation

With all the components in place, the final step is to calculate ROMI using the formula mentioned earlier:

$$\text{ROMI} = \frac{\text{Incremental Revenue} - \text{Marketing Cost}}{\text{Marketing Cost}} \times 100$$

Purpose of ROMI calculation:

- Evaluate the effectiveness of marketing campaigns.
- Optimize future marketing efforts based on insights from ROMI analysis.
- Make data-driven decisions about marketing spend and strategy.

Example: An e-commerce retailer calculates the incremental revenue generated by customers exposed to a marketing campaign and compares it with those who were not. By using the ROMI formula, the retailer determines the return on marketing investment and gains insights into the effectiveness of the campaign.

Advanced Techniques and Considerations

Advanced Attribution Models

Attribution modeling can be enhanced using advanced techniques to provide more accurate insights into the impact of marketing activities.

1. **Shapley Value Attribution:**
   - A cooperative game theory approach that fairly distributes the conversion value among all touchpoints.
   - It considers the marginal contribution of each touchpoint to the overall conversion, providing a more accurate representation of each channel's impact.

2. **Markov Chain Attribution:**
    - Uses transition probabilities to model the customer journey and attribute conversions accordingly.
    - It captures the probabilistic nature of customer interactions and provides insights into the most influential touchpoints in the conversion path.

Purpose:

- Improve the accuracy of attribution models.
- Gain deeper insights into the contribution of each marketing channel.
- Optimize marketing spend based on more precise attribution data.

Example: An e-commerce retailer uses Shapley Value Attribution to fairly distribute the conversion value among all touchpoints in a customer's journey. This approach provides a more accurate representation of each channel's impact, allowing the retailer to optimize budget allocation.

Machine Learning for Predictive Modeling

Machine learning models can be used to enhance predictive modeling for ROMI calculation. These models can capture complex relationships and provide more accurate forecasts.

Common machine learning models:

- **Neural Networks:** Capture complex, non-linear relationships in the data for more accurate predictions.
- **Ensemble Methods:** Combine multiple models to improve prediction accuracy and robustness. Examples include random forests and gradient boosting.

Purpose:

- Enhance the accuracy of predictive models.
- Capture complex relationships that traditional models may miss.
- Provide more reliable forecasts for incremental revenue and CLV.

Example: An e-commerce retailer uses a neural network to predict the incremental revenue generated by a marketing campaign. The model captures complex relationships in the data, providing more accurate forecasts and helping the retailer optimize marketing spend.

Real-Time Data Processing

Real-time data processing frameworks enable continuous updates to ROMI calculations based on the latest data. This allows businesses to make more timely and informed decisions.

Techniques:

- **Streaming Analytics:** Use real-time data processing frameworks such as Apache Kafka and Apache Spark Streaming to process data as it arrives.
- **Real-Time Dashboards:** Implement real-time dashboards to visualize ROMI and other key metrics, enabling quick decision-making.

Purpose:

- Provide up-to-date insights into the effectiveness of marketing campaigns.
- Enable timely adjustments to marketing strategies based on real-time data.
- Improve responsiveness to changes in customer behavior and market conditions.

Example: An e-commerce retailer implements a real-time dashboard to monitor ROMI and other key metrics continuously. By processing data in real-time, the retailer can quickly adjust marketing strategies based on the latest insights.

Experimentation and A/B Testing

Experimentation frameworks such as A/B testing help isolate the impact of specific marketing activities and provide more accurate estimates of incremental revenue.

Techniques:

- **A/B Testing:** Randomly assign customers to control and test groups to isolate the impact of specific marketing activities. Compare the outcomes between the two groups to estimate the incremental revenue generated by the marketing campaign.
- **Multi-Arm Bandit:** An adaptive experimentation framework that balances exploration and exploitation to optimize marketing strategies. It dynamically allocates resources to the most promising marketing activities based on real-time performance data.

Purpose:

- Isolate the impact of specific marketing activities.
- Provide more accurate estimates of incremental revenue.
- Optimize marketing strategies based on experimental results.

Example: An e-commerce retailer conducts an A/B test to evaluate the impact of a new email marketing campaign. Customers are randomly assigned to a control group (no exposure) and a test group (exposure to the campaign). By comparing the outcomes between the two groups, the retailer estimates the incremental revenue generated by the campaign and calculates ROMI.

ROMI Conclusion

Calculating Return on Marketing Investment (ROMI) using data science models involves a combination of techniques to accurately attribute revenue to marketing efforts, predict future revenue, and understand the long-term value of customers. By leveraging advanced models and methodologies, businesses can gain deeper insights into the effectiveness of their marketing campaigns and make data-driven decisions to optimize their marketing strategies.

In summary, the key steps in calculating ROMI using data science models include:

1. **Data Collection and Preparation:** Gather and prepare data from various sources to create a unified dataset.
2. **Customer Segmentation:** Divide customers into distinct groups to tailor marketing strategies and understand segment-specific responses.

3. **Attribution Modeling:** Determine the contribution of each marketing channel to conversions using advanced attribution models.
4. **Predictive Modeling for Incremental Revenue:** Forecast the revenue impact of marketing campaigns using machine learning models.
5. **Customer Lifetime Value (CLV) Analysis:** Understand the long-term value of customers and tailor marketing strategies to maximize CLV.
6. **Marketing Cost Calculation:** Accurately account for all marketing expenses to calculate the net profit generated by marketing activities.
7. **ROMI Calculation:** Use the ROMI formula to evaluate the effectiveness of marketing campaigns and make data-driven decisions about marketing spend.

By integrating these steps and leveraging advanced data science models, businesses can accurately calculate ROMI, optimize their marketing efforts, and ultimately drive higher returns on their marketing investments.

Product Engagement Scores

Product engagement scores are metrics used to quantify how users interact with a product. These scores provide insights into user behavior, product performance, and overall user satisfaction. They are essential for understanding the health of a product and guiding business decisions.

Importance of Product Engagement Scores

Product engagement scores are crucial for several reasons:

1. **User Retention:** High engagement often correlates with better user retention. Understanding engagement can help identify features that keep users coming back.

2. **Feature Improvement:** By analyzing engagement scores, companies can determine which features are most popular and which need improvement.
3. **Personalization:** Engagement scores can be used to personalize user experiences, tailoring content and features to individual user preferences.
4. **Marketing and Sales:** Engaged users are more likely to convert to paying customers. Engagement metrics can help refine marketing strategies.
5. **Product Development:** Insights from engagement scores can inform product development, ensuring that new features align with user needs and preferences.

Example 1: Streaming Platform

Metrics for engagement:

1. **Watch Time:** Total time spent watching content.
2. **Session Frequency:** Number of sessions per user over a specific period.
3. **Content Interaction:** Number of likes, comments, shares, and other interactions with content.
4. **Churn Rate:** Percentage of users who stop using the platform.
5. **Subscription Rate:** Percentage of users who convert to paid subscriptions.

Calculating the Engagement Score

Let's assume a streaming platform collects data on user interactions, including watch time, session frequency, and content interactions. You can create a composite engagement score by combining these metrics. Normalization typically involves scaling each metric to a common range, such as 0 to 1

Step 1: Normalize Metrics

First, you need to normalize the metrics to ensure they are on the same scale.

$$\text{Normalized Metric} = \frac{\text{Metric} - \min(\text{Metric})}{\max(\text{Metric}) - \min(\text{Metric})}$$

Step 2: Assign Weights

Next, you assign weights to each metric based on their importance. For example:

- Watch Time: 40%
- Session Frequency: 30%
- Content Interaction: 30%

Step 3: Calculate Composite Score

Finally, you calculate the composite engagement score.

$$\begin{aligned}\text{Engagement Score} = {} & 0.4 \times \text{Normalized Watch Time} + \\ & 0.3\text{x}\text{Normalized Session Frequency} + \\ & 0.3\text{x}\text{Normalized Content Interaction}\end{aligned}$$

Example calculation:
Assume the following normalized metrics for a user:

- Normalized Watch Time: 0.8
- Normalized Session Frequency: 0.6
- Normalized Content Interaction: 0.7

$$\begin{aligned}\text{Engagement Score} = {} & 0.4 \times 0.8 + 0.3\text{x}0.6 + \\ & 0.3\text{x}0.7 = 0.32 + 0.18 + 0.21 = 0.71\end{aligned}$$

Example 2: Software Application

Here are the metrics for engagement:

1. **Active Days:** Number of days the user interacts with the app.
2. **Feature Usage:** Number of unique features used by the user.
3. **Session Duration:** Average duration of a user session.
4. **Error Rate:** Frequency of encountering errors.
5. **Retention Rate:** Percentage of users returning after the first use.

Calculating the Engagement Score

For a software application, you can follow a similar approach to the streaming platform.

Step 1: Normalize Metrics

Normalize the metrics to ensure they are on the same scale.

$$\text{Normalized Metric} = \frac{\text{Metric} - \min(\text{Metric})}{\max(\text{Metric}) - \min(\text{Metric})}$$

Step 2: Assign Weights

Assign weights to each metric based on their importance. For example:

- Active Days: 25%
- Feature Usage: 25%
- Session Duration: 20%
- Error Rate: 10%
- Retention Rate: 20%

Step 3: Calculate Composite Score

Calculate the composite engagement score.

$$\text{Engagement Score} = 0.25\text{x}\,\text{Normalized Active Days} + 0.25\text{x}\,\text{Normalized Feature Usage} +$$

$$0.2\text{x}\,\text{Normalized Session Duration} + 0.1\text{x}(1 - \text{Normalized Error Rate}) +$$

$$0.2\text{x}\,\text{Normalized Retention Rate}$$

Note Error rate is subtracted from 1 because a higher error rate indicates lower engagement.

Example calculation:
Assume the following normalized metrics for a user:

- Normalized Active Days: 0.9
- Normalized Feature Usage: 0.7
- Normalized Session Duration: 0.8
- Normalized Error Rate: 0.2
- Normalized Retention Rate: 0.85

$$\text{Engagement Score} = 0.25\text{x}0.9 + 0.25\text{x}0.7 + 0.2\text{x}0.8 + 0.1\text{x}(1 - 0.2) + 0.2\text{x}\mathbf{0}.85][\text{Engagement Score} = 0.225 + 0.175 + 0.16 + 0.08 + 0.17 = 0.81$$

Careful Considerations

Error Rate Interpretation: Ensure that the interpretation of the error rate aligns with its impact on engagement.

Normalization Consistency: Use a consistent normalization method across all metrics, including the error rate.

Weight Adjustment: Adjust weights based on the relative importance of each metric and the impact of the error rate on overall engagement.

Validation: Validate the revised engagement score formula with real-world data to ensure its effectiveness in capturing user engagement.

Using Engagement Scores for Decision Making

Streaming platform:

1. **Content Recommendations:** Use engagement scores to recommend content that keeps users engaged.
2. **Churn Prediction:** Identify users with low engagement scores and target them with retention strategies.
3. **Marketing Campaigns:** Tailor marketing campaigns based on user engagement to maximize effectiveness.
4. **Product Development:** Focus on developing features and content that drive higher engagement.

Software application:

1. **Feature Prioritization:** Use engagement scores to identify which features are most important to users and prioritize their development.
2. **User Onboarding:** Enhance onboarding processes for users with low engagement scores to improve their experience.
3. **Bug Fixing:** Focus on reducing error rates for features with high engagement to maintain user satisfaction.
4. **Retention Strategies:** Develop targeted retention strategies for users with declining engagement scores.

Importance of Product Engagement Scores

1. **Customer Insights:** Engagement scores provide valuable insights into customer behavior and preferences, enabling businesses to make informed decisions.
2. **Product Improvement:** By identifying which features drive engagement, companies can focus their efforts on improving and expanding those features.
3. **Revenue Growth:** Higher engagement often leads to increased customer retention and conversion rates, driving revenue growth.
4. **Competitive Advantage:** Businesses that effectively monitor and respond to engagement metrics can stay ahead of competitors by continuously enhancing the user experience.
5. **Resource Allocation:** Engagement scores help allocate resources effectively by highlighting areas that need attention and investment.

Calculating product engagement scores using data science and mathematical techniques provides a quantitative measure of how users interact with a product. These scores are essential for understanding user behavior, guiding product development, and making informed business decisions. By leveraging engagement metrics, businesses can enhance user satisfaction, increase retention, and ultimately drive growth.

In summary, engagement scores for a streaming platform and a software application can be calculated using a combination of normalized metrics and weighted averages. These scores provide critical insights that can be used to personalize user experiences, optimize marketing strategies, improve product features, and allocate resources effectively. Understanding and utilizing engagement scores is crucial for any business aiming to thrive in today's competitive landscape.

Uplift Modeling

Uplift modeling, also known as incremental modeling or true lift modeling, is a data science technique used to predict the incremental impact of a treatment (such as a marketing campaign) on an individual's behavior. Unlike traditional predictive

modeling, which predicts an outcome for a given input, uplift modeling predicts the difference in outcomes between treated and untreated groups.

Importance of Uplift Modeling

1. **Targeted Marketing:** Uplift modeling helps identify individuals who are more likely to respond positively to a treatment, enabling more effective and efficient marketing strategies.
2. **Cost Efficiency:** By targeting only those who are likely to be influenced by the treatment, businesses can reduce marketing costs and increase return on investment.
3. **Customer Retention:** Uplift modeling can be used to identify customers who are at risk of churn and are likely to stay with the company if given an incentive.
4. **Personalization:** Uplift modeling enables personalized treatment strategies, improving customer satisfaction and loyalty.

The Theory of Uplift Modeling

Uplift modeling involves creating two models:

1. **Treatment Model:** Predicts the outcome for individuals who receive the treatment.
2. **Control Model:** Predicts the outcome for individuals who do not receive the treatment.

The difference between the predictions of these two models represents the uplift.

Basic Steps in Uplift Modeling

1. **Data Collection:** Gather data on treated and untreated individuals, including features that may influence their behavior.
2. **Feature Engineering:** Create relevant features that capture the characteristics of individuals.

3. **Model Building:** Build separate models for the treatment and control groups.
4. **Uplift Calculation:** Calculate the uplift by subtracting the predictions of the control model from the treatment model.
5. **Evaluation:** Evaluate the uplift model using appropriate metrics, such as a Qini coefficient or uplift curve.

In uplift modeling, individuals can be segmented into four key categories based on the potential impact of the treatment. These segments are often visualized using a two-by-two matrix (quadrants), where the axes represent the likelihood of a positive response with and without the treatment. The segments are as follows:

1. **Persuadables**
   - **Definition:** Individuals who are likely to respond positively to the treatment.
   - **Characteristics:** They would not take the desired action without the treatment but will do so with the treatment.
   - **Impact:** Positive uplift.
   - **Example:** Customers who will purchase the product only if they receive the promotional email.
2. **Sure Things**
   - **Definition:** Individuals who will take the desired action regardless of the treatment.
   - **Characteristics:** Their behavior does not change whether they receive the treatment or not.
   - **Impact:** No uplift (neutral).
   - **Example:** Customers who will purchase the product whether they receive the promotional email or not.
3. **Lost Causes**
   - **Definition:** Individuals who are unlikely to take the desired action regardless of the treatment.

- **Characteristics:** They will not respond positively whether they receive the treatment or not.
- **Impact:** No uplift (neutral).
- **Example:** Customers who will not purchase the product regardless of receiving the promotional email.

4. **Do Not Disturbs**
   - **Definition:** Individuals who might react negatively to the treatment.
   - **Characteristics:** They might have taken the desired action without the treatment but are less likely to do so because of the treatment.
   - **Impact:** Negative uplift.
   - **Example:** Customers who might be annoyed by the promotional email and decide not to purchase the product as a result.

Using the Uplift Quadrant Matrix

With Treatment	Positive Reponse	Minimal Response
	Persuadables	Sure Things
	Lost Causes	Do not Disturbs
	Negative Response	Avoid Response

Targeting strategy:

- **Persuadables:** Focus your resources on these individuals, as they provide the highest incremental gain.

- **Sure Things:** Do not spend resources on these individuals, as they will take the desired action any way.
- **Lost Causes:** Avoid targeting these individuals since they are unlikely to respond positively.
- **Do Not Disturbs:** Exclude these individuals from the treatment to avoid potential negative responses.

Practical Application

Let's say you are running a marketing campaign for a new product. After building and applying your uplift model, you find the following:

- **Persuadables:** 2,000 customers
- **Sure Things:** 1,500 customers
- **Lost Causes:** 3,000 customers
- **Do Not Disturbs:** 500 customers

By targeting the 2,000 persuadables, you maximize your marketing efficiency and increase the overall success of your campaign. Avoiding sure things, lost causes, and do not disturbs helps save costs and ensures a positive customer experience.

Example: Uplift Modeling for a Retail Business

Let's consider a retail business that wants to implement an uplift model to improve the effectiveness of a promotional campaign aimed at increasing customer purchases.

Fake data:

1. **Customer Data:** Includes customer demographics, purchase history, and engagement metrics.
2. **Treatment Data:** Indicates whether a customer received a promotional offer.

3. **Outcome Data:** Represents whether the customer made a purchase during the campaign period.

Here's a small sample of fake data:

CustomerID	Age	Gender	Purchase History	EngagementScore	Treatment	Purchase
1	25	F	5	80	1	1
2	34	M	2	60	0	0
3	45	F	10	90	1	1
4	30	M	3	50	0	1
5	50	F	8	85	1	0

- **CustomerID:** Unique identifier for each customer.
- **Age:** Age of the customer.
- **Gender:** Gender of the customer.
- **Purchase History:** Number of past purchases made by the customer.
- **Engagement Score:** A score representing the customer's engagement with the brand.
- **Treatment:** Indicates whether the customer received the promotional offer (1 for treated, 0 for control).
- **Purchase:** Indicates whether the customer made a purchase during the campaign period (1 for yes, 0 for no).

The Step-by-Step Uplift Modeling Process

Step 1: Data Collection and Preparation

Collect data on treated and untreated customers, including their demographics, purchase history, and engagement metrics. Ensure that the data is clean and consistent.

Step 2: Feature Engineering

Create features that capture the characteristics of customers. For example, you might create interaction terms between age and engagement score or between purchase history and treatment.

Step 3: Model Building

Build separate models for the treatment and control groups. These models predict the likelihood of a purchase for each group.

1. **Treatment Model:** Uses data from customers who received the promotional offer.
2. **Control Model:** Uses data from customers who did not receive the promotional offer.

Step 4: Uplift Calculation

Calculate the uplift for each customer by subtracting the predictions of the control model from the treatment model.

$$\text{Uplift} = \text{PredictionTreatment} - \text{PredictionControl}$$

Step 5: Evaluation

Evaluate the uplift model using metrics such as the Qini coefficient or uplift curve. These metrics measure the effectiveness of the model in identifying customers who are most likely to respond positively to the treatment.

Detailed Example

Step 1: Data Collection and Preparation

Assume you have collected data for 1,000 customers, including demographic information, purchase history, engagement scores, treatment status, and purchase outcomes.

Step 2: Feature Engineering

You create the following features:

- **Age:** Age of the customer.
- **Gender:** Gender of the customer (encoded as 0 for male and 1 for female).
- **Purchase History:** Number of past purchases.
- **Engagement Score:** Customer engagement score.
- **Treatment:** Binary variable indicating whether or not the customer received the promotional offer.

Step 3: Model Building

You split the data into treatment and control groups and build logistic regression models for each group.

Treatment Model:

$$\text{PurchaseProbability}_{\text{Treatment}} = \frac{1}{1+e^{-(\beta_0+\beta_1\cdot\text{Age}+\beta_2\cdot\text{Gender}+\beta_3\cdot\text{PurchaseHistory}+\beta_4\cdot\text{EngagementScore})}}$$

Control Model:

$$\text{PurchaseProbability}_{\text{Control}} = \frac{1}{1+e^{-(\alpha_0+\alpha_1\cdot\text{Age}+\alpha_2\cdot\text{Gender}+\alpha_3\cdot\text{PurchaseHistory}+\alpha_4\cdot\text{EngagementScore})}}$$

Step 4: Uplift Calculation

For each customer, you calculate the predicted purchase probabilities using both models and compute the uplift.

$$\text{Uplift} = \text{PurchaseProbabilityTreatment} - \text{PurchaseProbabilityControl}$$

Example Calculation:

For a customer with the following characteristics:

- Age: 30
- Gender: Male
- Purchase History: 3
- Engagement Score: 70

Assume the treatment model predicts a purchase probability of 0.6, and the control model predicts a purchase probability of 0.4.

$$\text{Uplift} = 0.6 - 0.4 = 0.2$$

Step 5: Evaluation

You evaluate the uplift model using the Qini coefficient, which measures the cumulative incremental response as a function of the cumulative number of customers targeted.

Qini curve:

- Plot the cumulative incremental response on the y-axis and the cumulative number of customers targeted on the x-axis.
- The Qini coefficient is the area between the Qini curve and the baseline.

Example: Email Marketing Campaign

Imagine you work for an online retail company, and you want to run an email marketing campaign to increase sales. You have a list of 10,000 customers and need to decide who should receive the email.

1. **Define the Treatment and Control Groups**
   - **Treatment Group:** 5,000 customers will receive the marketing email.
   - **Control Group:** 5,000 customers will not receive the marketing email.

2. **Run the Campaign**
    - Send the email to the treatment group.
    - Wait for a period (e.g., one month) and then measure the outcomes (e.g., whether the customers made a purchase).
3. **Collect Data**
    - Collect purchase data for the treatment and control groups.
    - Assume the data shows the following:
    - **Treatment Group:** 1,000 out of 5,000 customers made a purchase.
    - **Control Group:** 800 out of 5,000 customers made a purchase.
4. **Calculate the Uplift**
    - Uplift is the difference in response rates between the treatment and control groups.
    - Response rate in the treatment group: 1000/5000 = 20%
    - Response rate in the control group: 800/5000 = 16%
    - Uplift: 20% - 16% = 4%
5. **Develop the Uplift Model**
    - Use machine learning techniques (e.g., decision trees, random forests) to build an uplift model that predicts the incremental effect of the email on each customer.
    - The model will use features such as past purchase behavior, demographic information, browsing history, and so on, to estimate the uplift for each customer.
6. **Segment the Customers**
    - **Persuadables:** Customers predicted to have a positive uplift (more likely to purchase because of the email).
    - **Sure Things:** Customers predicted to purchase regardless of receiving the email.

- **Lost Causes:** Customers predicted not to purchase regardless of receiving the email.
- **Do Not Disturbs:** Customers predicted to have a negative response (less likely to purchase because of the email).

7. **Optimize the Campaign**
   - Focus the email campaign on the persuadables to maximize the return on investment (ROI).
   - Avoid sending emails to sure things (no additional benefit), lost causes (wasted effort), and do not disturbs (potential negative impact).

Benefits of Uplift Modeling

- **Targeted Marketing:** By identifying persuadables, you can tailor your marketing efforts to those who are most likely to be influenced, improving the efficiency of your campaigns.
- **Cost Savings:** Reducing waste by not targeting sure things, lost causes, and do not disturbs saves resources and minimizes potential negative impacts.
- **Improved Customer Experience:** Avoiding unnecessary or potentially annoying communications helps maintain a positive relationship with your customers.

In summary, uplift modeling helps companies focus their marketing efforts on individuals who are most likely to be positively influenced by their campaigns, leading to better outcomes and more efficient use of resources.

Uplift modeling is a powerful technique that helps businesses optimize their marketing strategies by targeting individuals who are most likely to respond positively to a treatment. By leveraging data science and machine learning models, businesses can accurately predict the incremental impact of their marketing efforts, leading to more effective and cost-efficient campaigns.

In the example of a retail business, we demonstrated how to collect and prepare data, create relevant features, build treatment and control models, calculate uplift, and evaluate the model using the Qini coefficient. This comprehensive approach enables businesses to make data-driven decisions, improve customer engagement, and maximize return on investment.

Summary

In summary, this chapter discussed various frameworks, including these:

1. **Customer Lifetime Value (CLV)**

   Customer Lifetime Value (CLV) is a critical metric that estimates the total value a customer brings to a business over their entire relationship. By predicting CLV, companies can identify high-value customers and tailor their marketing efforts accordingly.

2. **Marketing Framework for Return on Marketing Investment (ROMI)**

   Effective marketing is essential for business growth, but understanding the return on marketing investment (ROMI) is equally crucial. Predictive models can be used to measure ROMI, helping businesses allocate their marketing budgets more efficiently. By stacking predictive models, companies can forecast the impact of marketing campaigns, optimize spend, and maximize returns.

3. **Measuring Product Engagement**

   Product engagement metrics provide insights into how customers interact with a product, which is vital for product development and customer satisfaction. By understanding these metrics, businesses can enhance user experience and drive product improvements.

4. **Uplift Modeling**

   Uplift modeling focuses on identifying individuals who can be positively influenced by specific interventions. By predicting the incremental impact of marketing efforts, businesses can target customers who are most likely to respond favorably, thereby improving retention rates and maximizing marketing efficiency.

Conclusion

By stacking predictive models into comprehensive frameworks, businesses can unlock deeper insights and create more effective strategies. This chapter aimed to provide a thorough understanding of how these models work together to add value, offering practical guidance for leveraging predictive analytics to achieve business objectives. Through the exploration of customer lifetime value, marketing investment returns, product engagement metrics, and uplift modeling frameworks, you will gain the knowledge needed to apply these powerful tools in their own organizations.